

How to use the PowerShell documentation

Article • 01/17/2025

Welcome to the PowerShell online documentation. This site contains cmdlet reference for the following versions of PowerShell:

- PowerShell 7.6 (preview)
- PowerShell 7.5 (RC)
- PowerShell 7.4 (LTS)
- PowerShell 5.1

Navigating the documentation

The screenshot shows a web browser displaying the Microsoft Learn - PowerShell website. The URL is <https://learn.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7.3>. The page title is "What is PowerShell?". The left sidebar has a "Version" dropdown set to "PowerShell 7.3" and a "Filter by title" input field. A search bar at the top right contains the placeholder "Search". The main content area features the article "What is PowerShell?", which includes a brief description of PowerShell as a cross-platform task automation solution and a "Command-line Shell" section detailing its features like history, tab completion, and pipelines. To the right, there's a sidebar titled "In this article" with links to "Command-line Shell", "Scripting language", "Automation platform", and "Next steps".

The web page contains multiple elements that help you navigate the documentation.

- **Site level navigation** - The site level navigation appears at the top of the page. It contains links to other content on the Microsoft Learn platform.

- **Related content navigation** - The related content bar is immediately below the site level navigation. It contains links to content related to the current documentation set, which is PowerShell in this case.
- **Version selector** - The version selector appears above the Table of Contents (TOC) and controls which version of the cmdlet reference appears in the TOC.
- **Table of Contents** - The TOC on the left side of the page is divided into two sections: conceptual and reference. Notice the line between the **Reference** node of the TOC. The conceptual documents appear above the line. Reference content is listed in **Reference** node below the line.
- **Action buttons** - The action buttons provide a way to add content to a collection, provide feedback, edit the content, or share the content with others.

Selecting the version of PowerShell

Use the version selector located above the TOC to select the version of PowerShell you want. By default, the page loads with the most current stable release version selected. The version selector controls which version of the cmdlet reference appears in the TOC under the **Reference** node. Some cmdlets work differently in different versions of PowerShell you are using. Be sure you are viewing the documentation for the correct version of PowerShell.

The version selector doesn't affect conceptual documentation. The conceptual documents appear above the **Reference** node in the TOC. The same conceptual articles appear for every version selected. If there are version-specific differences, the documentation makes note of those differences.

The screenshot shows a Microsoft Docs page for 'PowerShell Scripting'. The URL in the address bar is docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-6. The page title is 'PowerShell'. The sidebar on the left has a 'Version' dropdown set to 'PowerShell 6', with a 'Filter by title' input field below it. Under 'How to use this documentation', the 'Overview' section is expanded, showing links like 'Install', 'Get started', 'Learning PowerShell', etc. The main content area starts with a heading 'PowerShell' and a date '08/26/2018 • 2 minutes to read'. It describes PowerShell as a task-based command-line shell and scripting language built on .NET, used for managing operating systems. A 'PowerShell is open-source' section is present at the bottom, mentioning GitHub and DSC.

You can verify the version of PowerShell you are using by inspecting the `$PSVersionTable.PSVersion` value. The following example shows the output for Windows PowerShell 5.1.

```
PowerShell  
$PSVersionTable.PSVersion  
  
Output  
Major  Minor  Build  Revision  
-----  -----  -----  -----  
5       1       22621  963
```

Finding articles

There are two ways to search for content in Docs.

- The search box in the site-level navigation bar searches the entire site. It returns a list of matching articles from all documentation sets.
- The TOC filter box under the version selector allows filtering by words that appear in the title of an article. The filter displays a list of matching articles as you type. You can also select the option to search for the words in an article. When you search from here, the search is limited to the PowerShell documentation.

In the following example, the search in the site-level navigation bar returns 840 results for the word `idempotent`. Entering the word `invoke` in the TOC filter box shows a list of articles that contain the word `invoke` in the title. Entering the word `idempotent` in the TOC filter shows no articles. Clicking the search link searches for `idempotent` in the PowerShell documentation. This search only returns 9 results.

The screenshot shows a Microsoft Edge browser window displaying the Microsoft Learn - PowerShell documentation. The URL is <https://learn.microsoft.com/en-us/powershell/scripting/overview?...>. The page title is "What is PowerShell? - PowerShell". The left sidebar has a "Version" dropdown set to "PowerShell 7.3" and a "Filter by title" input field. The main content area shows the "What is PowerShell?" article, which includes a brief introduction, a "Command-line Shell" section, and a "Scripting language" section. To the right, there's a "Feedback" button and a "Additional resources" sidebar with sections for "Training", "Documentation", and "Discover PowerShell". A "Download PDF" button is located at the top right of the main content area.

Downloading the documentation as a PDF

To download the documentation as a PDF, click the **Download PDF** button at the bottom of the TOC.

Version

PowerShell 7.3

[Filter by title](#)

How to use this documentation

> Overview

> Install

> Learning PowerShell

> What's New in PowerShell

> Windows PowerShell

Desired State Configuration (DSC)

PowerShell Gallery

> Community

> Scripting and development

> Docs Contributor's Guide

PowerShell support lifecycle

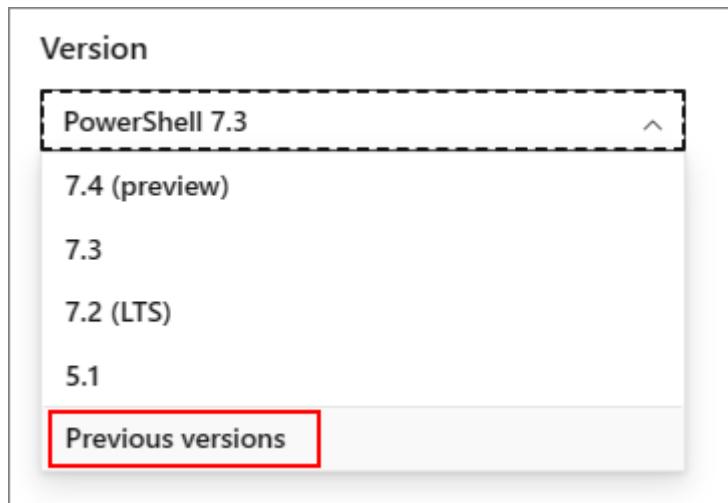
> Reference

[Download PDF](#)

- If you are viewing a conceptual article, the Learn platform creates a PDF containing all the conceptual content for the selected version.
- If you are viewing a reference article, the Learn platform creates a PDF containing all the reference content for the selected version.

Finding articles for previous versions

Documentation for older versions of PowerShell is archived in our [Previous Versions](#) site. You can choose **Previous Versions** from the version selector.



The previous versions site contains documentation for the following topics:

- PowerShell 7.3
- PowerShell 7.2
- PowerShell 7.1
- PowerShell 7.0
- PowerShell 6
- PowerShell 5.0
 - PowerShell Workflows
 - PowerShell Web Access
- PowerShell 4.0
- PowerShell 3.0

What is PowerShell?

Article • 10/30/2024

PowerShell is a cross-platform task automation solution made up of a command-line shell, a scripting language, and a configuration management framework. PowerShell runs on Windows, Linux, and macOS.

Command-line Shell

PowerShell is a modern command shell that includes the best features of other popular shells. Unlike most shells that only accept and return text, PowerShell accepts and returns .NET objects. The shell includes the following features:

- Robust command-line [history](#)
- Tab completion and command prediction (See [about_PSReadLine](#))
- Supports command and parameter [aliases](#)
- [Pipeline](#) for chaining commands
- In-console [help](#) system, similar to Unix `man` pages

Scripting language

As a scripting language, PowerShell is commonly used for automating the management of systems. It's also used to build, test, and deploy solutions, often in CI/CD environments. PowerShell is built on the .NET Common Language Runtime (CLR). All inputs and outputs are .NET objects. No need to parse text output to extract information from output. The PowerShell scripting language includes the following features:

- Extensible through [functions](#), [classes](#), [scripts](#), and [modules](#)
- Extensible [formatting system](#) for easy output
- Extensible [type system](#) for creating dynamic types
- Built-in support for common data formats like [CSV](#), [JSON](#), and [XML](#)

Automation platform

The extensible nature of PowerShell has enabled an ecosystem of PowerShell modules to deploy and manage almost any technology you work with. For example:

Microsoft

- [Azure](#)

- Windows
- Exchange
- SQL

Third-party

- AWS ↗
- VMware ↗
- Google Cloud ↗

Configuration management

PowerShell Desired State Configuration ([DSC](#)) is a management framework in PowerShell that enables you to manage your enterprise infrastructure with configuration as code. With DSC, you can:

- Create declarative [configurations](#) and custom scripts for repeatable deployments
- Enforce configuration settings and report on configuration drift
- Deploy configuration using [push or pull](#) models

Next steps

Getting started

Are you new to PowerShell and don't know where to start? Take a look at these resources.

- [Installing PowerShell](#)
- [Discover PowerShell](#)
- [PowerShell 101](#)
- [Microsoft Virtual Academy videos](#)
- [PowerShell Learn modules](#)

PowerShell in action

Take a look at how PowerShell is being used in different scenarios and on different platforms.

- [PowerShell remoting over SSH](#)
- [Getting started with Azure PowerShell](#)
- [Building a CI/CD pipeline with DSC](#)

- Managing Microsoft Exchange

What is Windows PowerShell?

Article • 03/07/2024

Windows PowerShell and *PowerShell* are two separate products.

- *Windows PowerShell* is the version of PowerShell that ships in Windows. This version of PowerShell uses the full .NET Framework, which only runs on Windows. The latest version is Windows PowerShell 5.1. Microsoft is no longer updating Windows PowerShell with new features. Support for Windows PowerShell is tied to the version of Windows you are using.
- *PowerShell* is built on the new versions of .NET instead of the .NET Framework and runs on Windows, Linux, and macOS. Support for PowerShell is based on the version of .NET that it was built on. For more information about the support lifecycle for PowerShell, see the [PowerShell support lifecycle](#) documentation.

Further reading

- For a more detailed explanation of the differences between Windows PowerShell and PowerShell, see [Differences between Windows PowerShell 5.1 and PowerShell 7.x](#).
- For information about migrating from Windows PowerShell to PowerShell, see [Migrating from Windows PowerShell 5.1 to PowerShell 7](#).
- For more information about previous versions of Windows PowerShell, see [Previous versions of PowerShell](#).
- For more information about the terminology used in PowerShell documentation, see [Product terminology and branding guidelines](#).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

What is a command shell?

Article • 02/13/2025

Many people use the terms *command shell*, *command-line tool*, and *terminal* interchangeably, which can be confusing. This article explains the difference between these concepts and provides examples of each.

A **command shell** is an interactive command-line interface for managing a computer, also known as a **Read-Eval-Print Loop** ([REPL ↗](#)).

A shell takes input from the keyboard, evaluates that input, and executes the input as a shell command or forwards the input to the operating system to be executed. Most shells can also read commands from a script file, and can include programming features like variables, flow control, and functions.

Types of command shells

There are two main types of command shells:

- General purpose command shells

General purpose command shells provide are designed to work with the operating system and allow you to run any command that the operating system supports.

They also include shell-specific commands and programming features. The following list contains some examples of general purpose command shells:

- [PowerShell](#)
- [Windows Command Shell](#)
- [bash ↗](#) - popular on Linux
- [zsh ↗](#) - popular on macOS

- Utility command shells

Utility command shells are designed to work with specific applications or services. These shells can only run commands that are specific to the application or service. Some utility shells support running commands from a batch script, but don't include programming features. Usually, these shells can only be used interactively.

- [AI Shell](#) - An interactive-only shell used to communicate with AI services such as Azure OpenAI.
- [netsh](#) - Network shell (netsh) is a command-line utility that allows you to configure and display the status of various network components on Windows.

It's both a command-line tool and a command shell. It also supports running commands from a script file.

Command-line tools

A **command-line tool** is a standalone program that you run from a command shell. Command-line tools are typically designed to perform a specific task, such as managing files, configuring settings, or querying for information. Command-line tools can be used in any shell that supports running external programs.

- [Azure CLI](#) - a collection of command-line tools for managing Azure resources that can be run in any supported shell.
- [Azure PowerShell](#) - a collection of PowerShell modules for managing Azure resources that can be run in any supported version of PowerShell.
- [OpenSSH for Windows](#) - includes a command-line client and a server that provides secure communication over a network.
- [Windows Commands](#) - a collection of command-line tools that are built into Windows.

In general, command-line tools don't provide a command shell (REPL) interface. The `netsh` command in Windows is an exception, as it's both a command-line tool and an interactive command shell.

Terminals

A **terminal** is an application that provides a text-based window for hosting command shells. Some terminals are designed to work with a specific shell, while others can host multiple shells. They can also include advanced features such as:

- Ability to create multiple panes within a single window
- Ability to create multiple tabs to host multiple shells
- Ability to change color schemes and fonts
- Support for copy and paste operations

The following list contains some examples of terminal applications:

- [Windows Terminal](#) - a modern terminal application for Windows that can host multiple shells.
- [Windows Console Host](#) - the default host application on Windows for text-based applications. It can also host the Windows Command Shell or PowerShell.
- [Terminal for macOS](#) - the default terminal application on macOS that can host the bash or zsh shell.

- [iTerm2 for macOS](#) - a popular 3rd-party terminal application for macOS.
- [Azure Cloud Shell](#) - a browser-based terminal application hosted in Microsoft Azure. Azure Cloud shell gives you the choice of using bash or PowerShell. Each shell comes preconfigured with many command-line tools for managing Azure resources.

What is a PowerShell command (cmdlet)?

Article • 03/07/2024

Commands for PowerShell are known as cmdlets (pronounced command-lets). In addition to cmdlets, PowerShell allows you to run any command available on your system.

What is a cmdlet?

Cmdlets are native PowerShell commands, not stand-alone executables. Cmdlets are collected into PowerShell modules that can be loaded on demand. Cmdlets can be written in any compiled .NET language or in the PowerShell scripting language itself.

Cmdlet names

PowerShell uses a *Verb-Noun* name pair to name cmdlets. For example, the `Get-Command` cmdlet included in PowerShell is used to get all the cmdlets that are registered in the command shell. The verb identifies the action that the cmdlet performs, and the noun identifies the resource on which the cmdlet performs its action.

Next steps

To learn more about PowerShell and how to find other cmdlets, see the PowerShell Bits tutorial [Discover PowerShell](#).

For more information about creating your own cmdlets, see the following resources:

Script-based cmdlets

- [about_Functions_Advanced](#)
- [about_Functions_CmdletBindingAttribute](#)
- [about_Functions_Advanced_Methods](#)

Compiled cmdlets (PowerShell SDK docs)

- [Cmdlet overview](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Discover PowerShell

Article • 03/07/2024

PowerShell is a command-line shell and a scripting language in one. PowerShell started out on Windows to help automate administrative tasks. Now, it runs cross platform and can be used for various tasks.

The thing that makes PowerShell unique is that it accepts and returns .NET objects, rather than text. This feature makes it easier to connect different commands in a *pipeline*.

What can PowerShell be used for?

Usage of PowerShell has grown since the days when it was Windows-only. It's still used for Windows task automation, but today, you can use it for tasks like:

- **Cloud management.** PowerShell can be used to manage cloud resources. For example, you can retrieve information about cloud resources, as well as update or deploy new resources.
- **CI/CD.** It can also be used as part of a Continuous Integration/Continuous Deployment pipeline.
- **Automate tasks for Active Directory and Exchange.** You can use it to automate almost any task on Windows like creating users in Active Directory and mailboxes in Exchange.

There are many more areas of usage but the preceding list gives you a hint that PowerShell has come a long way.

Who uses PowerShell?

PowerShell is a powerful tool that can help people working in a multitude of roles. Traditionally, PowerShell has been used by the System Administrator role but is now being used by people calling themselves DevOps, Cloud Ops, and even Developers.

PowerShell cmdlets

PowerShell comes with hundreds of preinstalled commands. PowerShell commands are called cmdlets (pronounced *command-lets*).

The name of each cmdlet consists of a *Verb-Noun* pair. For example, `Get-Process`. This naming convention makes it easier to understand what the cmdlet does. It also makes it easier to find the command you're looking for. When looking for a cmdlet to use, you can filter on the verb or noun.

Using cmdlets to explore PowerShell

When you first pick up PowerShell, it might feel intimidating as there seems to be so much to learn. PowerShell is designed to help you learn a little at a time, as you need it.

PowerShell includes cmdlets that help you discover PowerShell. Using these three cmdlets, you can discover what commands are available, what they do, and what types they operate on.

- `Get-Verb`. Running this command returns a list of verbs that most commands adhere to. The response includes a description of what these verbs do. Since most commands follow this naming convention, it sets expectations on what a command does. This helps you select the appropriate command and what to name a command, should you be creating one.
- `Get-Command`. This command retrieves a list of all commands installed on your machine.
- `Get-Member`. It operates on object based output and is able to discover what object, properties and methods are available for a command.
- `Get-Help`. Invoking this command with the name of a command as an argument displays a help page describing various parts of a command.

Using these commands, you can discover almost anything you need to know about PowerShell.

Verb

Verb is an important concept in PowerShell. It's a naming standard that most cmdlets follow. It's also a naming standard you're expected to follow when you write your own commands. The idea is that the *Verb* says what you're trying to do, like read or maybe change data. PowerShell has a standardized list of verbs. To get a full list of all possible verbs, use the `Get-Verb` cmdlet:

```
PowerShell
Get-Verb
```

The cmdlet returns a long list of verbs. The **Description** provides context for what the verb is meant to do. Here's the first few rows of output:

Output			
Verb	AliasPrefix	Group	Description
Add	a	Common	Adds a resource to a container, or attaches an item to another item
Clear	cl	Common	Removes all the resources from a container but does not delete the container
Close	cs	Common	Changes the state of a resource to make it inaccessible, unavailable, or unusab...
Copy	cp	Common	Copies a resource to another name or to another container
Enter	et	Common	Specifies an action that allows the user to move into a resource
Exit	ex	Common	Sets the current environment or context to the most recently used context
...			

Find commands with Get-Command

The `Get-Command` cmdlet returns a list of all available commands installed on your system. The list you get back is quite large. You can limit the amount of information that comes back by filtering the response using parameters or helper cmdlets.

Filter on name

You can filter the output of `Get-Command` using different parameters. Filtering allows you to find commands that have certain properties. The **Name** parameter allows you to find a specific command by name.

PowerShell			
Get-Command -Name Get-Process			
Output			
CommandType	Name	Version	Source
Cmdlet	Get-Process	7.0.0.0	Microsoft.PowerShell.Management

What if you want to find all the commands that work with processes? You can use a wildcard `*` to match other forms of the string. For example:

PowerShell

```
Get-Command -Name *-Process
```

Output

CommandType	Name	Version	Source
Cmdlet	Debug-Process	7.0.0.0	Microsoft.PowerShell.Management
Cmdlet	Get-Process	7.0.0.0	Microsoft.PowerShell.Management
Cmdlet	Start-Process	7.0.0.0	Microsoft.PowerShell.Management
Cmdlet	Stop-Process	7.0.0.0	Microsoft.PowerShell.Management
Cmdlet	Wait-Process	7.0.0.0	Microsoft.PowerShell.Management

Filtering on Noun and Verb

There are other parameters that filter on verb and noun values. The verb part of a command's name is the leftmost part. The verb should be one of the values returned by the `Get-Verb` cmdlet. The rightmost part of a command is the noun part. A noun can be anything.

- **Filter on verb.** In the command `Get-Process`, the verb part is `Get`. To filter on the verb part, use the `Verb` parameter.

PowerShell

```
Get-Command -Verb 'Get'
```

This example lists all commands that use the verb `Get`.

- **Filter on noun.** In the command `Get-Process`, the noun part is `Process`. To filter on the noun, use the `Noun` parameter. The following example returns all cmdlets that have nouns starting with the letter `U`.

PowerShell

```
Get-Command -Noun U*
```

Also, you can combine parameters to narrow down your search, for example:

PowerShell

```
Get-Command -Verb Get -Noun U*
```

Output

CommandType	Name	Version	Source
Cmdlet	Get-UICulture	7.0.0.0	
Microsoft.PowerShell.Utility			
Cmdlet	Get-Unique	7.0.0.0	
Microsoft.PowerShell.Utility			
Cmdlet	Get-Uptime	7.0.0.0	
Microsoft.PowerShell.Utility			

Use helper cmdlets to filter results

You can also use other cmdlets to filter results.

- `Select-Object`. This versatile command helps you pick out specific properties from one or more objects. You can also limit the number of items you get back. The following example returns the **Name** and **Source** property values for the first 5 commands available in the current session.

PowerShell

```
Get-Command | Select-Object -First 5 -Property Name, Source
```

Output

Name	Source
Add-AppPackage	Appx
Add-AppPackageVolume	Appx
Add-AppProvisionedPackage	Dism
Add-AssertionOperator	Pester
Add-ProvisionedAppPackage	Dism

For more information, see [Select-Object](#).

- `Where-Object`. This cmdlet lets you filter the objects returned based on the values of properties. The command takes an expression that can test the value of a property. The following example returns all processes where the `ProcessName` starts with `p`.

PowerShell

```
Get-Process | Where-Object {$_.ProcessName -like "p*"}  
The Get-Process cmdlet returns a collection of process objects. To filter the response, pipe the output to Where-Object. Piping means that two or more commands are connected via a pipe | character. The output from one command is sent as the input for the next command. The filter expression for Where-Object uses the -like operator to match processes that start with the letter p.
```

Explore objects with Get-Member

Once you've been able to locate the cmdlet you want, you want to know more about what output it produces. The Get-Member cmdlet displays the type, properties, and methods of an object. Pipe the output you want to inspect to Get-Member.

PowerShell

```
Get-Process | Get-Member
```

The result displays the returned type as TypeName and all the properties and methods of the object. Here's an excerpt of such a result:

Output

Name	MemberType	Definition
Handles	AliasProperty	Handles = Handlecount
Name	AliasProperty	Name = ProcessName
...		

Using the MemberType parameter you can limit the information returned.

PowerShell

```
Get-Process | Get-Member -MemberType Method
```

By default PowerShell only displays a few properties. The previous example displayed the Name, MemberType and Definition members. You can use Select-Object to specify

properties you want to see. For example, you want to display only the `Name` and `Definition` properties:

```
PowerShell
```

```
Get-Process | Get-Member | Select-Object Name, Definition
```

Search by parameter type

`Get-Member` showed us that `Get-Process` returns `Process` type objects. The `ParameterType` parameter of `Get-Command` can be used to find other commands that take `Process` objects as input.

```
PowerShell
```

```
Get-Command -ParameterType Process
```

```
Output
```

CommandType	Name	Version	Source
Cmdlet	Debug-Process	7.0.0.0	
Microsoft.PowerShell.Managem...			
Cmdlet	Enter-PSThread	7.1.0.0	
Microsoft.PowerShell.Core			
Cmdlet	Get-Process	7.0.0.0	
Microsoft.PowerShell.Managem...			
Cmdlet	Get-PSThreadInfo	7.1.0.0	
Microsoft.PowerShell.Core			
Cmdlet	Stop-Process	7.0.0.0	
Microsoft.PowerShell.Managem...			
Cmdlet	Wait-Process	7.0.0.0	
Microsoft.PowerShell.Managem...			

Knowing the output type of a command can help narrow down your search for related commands.

Additional resources

- [Get-Command](#)
- [Get-Member](#)
- [Select-Object](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install PowerShell on Windows, Linux, and macOS

Learn about installing PowerShell on Windows, Linux, and macOS.

Windows



OVERVIEW

[Install PowerShell on Windows](#)

[Supported Windows releases](#)

macOS



OVERVIEW

[Install on macOS](#)

[Supported macOS releases](#)

Linux



OVERVIEW

[Linux overview](#)

[Alpine](#)

[Debian](#)

[Red Hat Enterprise Linux](#)

[Ubuntu](#)

Q&A



GET STARTED

[Alternate install methods](#)

[Community supported Linux](#)

[Using PowerShell in Docker](#)

[Arm Processor support](#)

[Microsoft Update FAQ for PowerShell](#)

[PowerShell Support Lifecycle](#)

Installing PowerShell on Windows

Article • 04/28/2025

There are multiple ways to install PowerShell in Windows. Each install method is designed to support different scenarios and workflows. Choose the method that best suits your needs.

- [WinGet](#) - Recommended way to install PowerShell on Windows clients
- [MSI package](#) - Best choice for Windows Servers and enterprise deployment scenarios
- [ZIP package](#) - Easiest way to "side load" or install multiple versions
 - Use this method for Windows Nano Server, Windows IoT, and Arm-based systems
- [.NET Global tool](#) - A good choice for .NET developers that install and use other global tools
- [Microsoft Store package](#) - An easy way to install for casual users of PowerShell but has limitations

ⓘ Note

The installation commands in this article are for the latest stable release of PowerShell. To install a different version of PowerShell, adjust the command to match the version you need. The following links direct you to the release page for each version in the PowerShell repository on GitHub.

- v7.5.0 - Stable release: <https://aka.ms/powershell-release?tag=stable>
- v7.4.7 - LTS release: <https://aka.ms/powershell-release?tag=lts>
- v7.6.0-preview.2 - Preview release: <https://aka.ms/powershell-release?tag=preview>

Download links for every package are found in the **Assets** section of the Release page. The **Assets** section may be collapsed, so you may need to click to expand it.

Install PowerShell using WinGet (recommended)

WinGet, the Windows Package Manager, is a command-line tool enables users to discover, install, upgrade, remove, and configure applications on Windows client computers. This tool is the client interface to the Windows Package Manager service. The `winget` command-line tool is bundled with Windows 11 and modern versions of Windows 10 by default as the **App Installer**.

ⓘ Note

See the [winget documentation](#) for a list of system requirements and install instructions. `winget` isn't available on Windows Server 2022 or earlier versions. Windows Server 2025

Preview Build 26085 and later includes winget for Windows Server with Desktop Experience only.

The following commands can be used to install PowerShell using the published winget packages:

Search for the latest version of PowerShell

PowerShell

```
winget search Microsoft.PowerShell
```

Output

Name	Id	Version	Source
<hr/>			
PowerShell	Microsoft.PowerShell	7.5.1.0	winget
PowerShell Preview	Microsoft.PowerShell.Preview	7.6.0.4	winget

Install PowerShell or PowerShell Preview using the `id` parameter

PowerShell

```
winget install --id Microsoft.PowerShell --source winget
```

PowerShell

```
winget install --id Microsoft.PowerShell.Preview --source winget
```

ⓘ Note

On Windows systems using X86 or X64 processor, winget installs the MSI package. On systems using the Arm64 processor, winget installs the Microsoft Store (MSIX) package. For more information, see [Installing from the Microsoft Store](#).

Installing the MSI package

To install PowerShell on Windows, use the following links to download the install package from GitHub.

- [PowerShell-7.5.1-win-x64.msi ↗](#)

- [PowerShell-7.5.1-win-x86.msi](#)
- [PowerShell-7.5.1-win-arm64.msi](#)

Once downloaded, double-click the installer file and follow the prompts.

The installer creates a shortcut in the Windows Start Menu.

- By default the package is installed to `$Env:ProgramFiles\PowerShell\<version>`
- You can launch PowerShell via the Start Menu or `$Env:ProgramFiles\PowerShell\<version>\pwsh.exe`

Note

PowerShell 7.4 installs to a new directory and runs side-by-side with Windows PowerShell 5.1. PowerShell 7.4 is an in-place upgrade that removes previous versions of PowerShell 7. Preview versions of PowerShell can be installed side-by-side with other versions of PowerShell.

- PowerShell 7.4 is installed to `$Env:ProgramFiles\PowerShell\7`
- The `$Env:ProgramFiles\PowerShell\7` folder is added to `$Env:PATH`

If you need to run PowerShell 7.4 side-by-side with other versions, use the [ZIP install](#) method to install the other version to a different folder.

Support for Microsoft Update in PowerShell 7.2 and newer

PowerShell 7.2 and newer has support for Microsoft Update. When you enable this feature, you'll get the latest PowerShell 7 updates in your traditional Microsoft Update (MU) management flow, whether that's with Windows Update for Business, WSUS, Microsoft Endpoint Configuration Manager, or the interactive MU dialog in Settings.

The PowerShell MSI package includes following command-line options:

- `USE_MU` - This property has two possible values:
 - `1` (default) - Opts into updating through Microsoft Update, WSUS, or Configuration Manager
 - `0` - Don't opt into updating through Microsoft Update, WSUS, or Configuration Manager
- `ENABLE_MU`
 - `1` (default) - Opts into using Microsoft Update for Automatic Updates
 - `0` - Don't opt into using Microsoft Update

Note

Enabling updates may have been set in a previous installation or manual configuration. Using `ENABLE_MU=0` doesn't remove the existing settings. Also, this setting can be overruled by Group Policy settings controlled by your administrator.

For more information, see the [PowerShell Microsoft Update FAQ](#).

Install the MSI package from the command line

MSI packages can be installed from the command line allowing administrators to deploy packages without user interaction. The MSI package includes the following properties to control the installation options:

- `ADD_EXPLORER_CONTEXT_MENU_OPENPOWERSHELL` - This property controls the option for adding the `Open PowerShell` item to the context menu in Windows Explorer.
- `ADD_FILE_CONTEXT_MENU_RUNPOWERSHELL` - This property controls the option for adding the `Run with PowerShell` item to the context menu in Windows Explorer.
- `ENABLE_PSREMOTING` - This property controls the option for enabling PowerShell remoting during installation.
- `REGISTER_MANIFEST` - This property controls the option for registering the Windows Event Logging manifest.
- `ADD_PATH` - This property controls the option for adding PowerShell to the Windows PATH environment variable.
- `DISABLE_TELEMETRY` - This property controls the option for disabling PowerShell's telemetry by setting the `POWERSHELL_TELEMETRY_OPTOUT` environment variable.
- `INSTALLFOLDER` - This property controls the installation directory. The default is `$Env:ProgramFiles\PowerShell\`. This is the location where the installer creates the versioned subfolder. You can't change the name of the versioned subfolder.
 - For current releases, the versioned subfolder is `7`
 - For preview releases, the versioned subfolder is `7-preview`

The following example shows how to silently install PowerShell with all the install options enabled.

PowerShell

```
msiexec.exe /package PowerShell-7.5.1-win-x64.msi /quiet  
ADD_EXPLORER_CONTEXT_MENU_OPENPOWERSHELL=1 ADD_FILE_CONTEXT_MENU_RUNPOWERSHELL=1  
ENABLE_PSREMOTING=1 REGISTER_MANIFEST=1 USE_MU=1 ENABLE_MU=1 ADD_PATH=1
```

For a full list of command-line options for `Msiexec.exe`, see [Command line options](#).

Installing the ZIP package

PowerShell binary ZIP archives are provided to enable advanced deployment scenarios.

Download one of the following ZIP archives from the [current release](#) page.

- [PowerShell-7.5.1-win-x64.zip](#)
- [PowerShell-7.5.1-win-x86.zip](#)
- [PowerShell-7.5.1-win-arm64.zip](#)

Depending on how you download the file you may need to unblock the file using the `Unblock-File` cmdlet. Unzip the contents to the location of your choice and run `pwsh.exe` from there.

Unlike installing the MSI packages, installing the ZIP archive doesn't check for prerequisites. For remoting over WSMAN to work properly, ensure that you've met the [prerequisites](#).

Use this method to install the ARM-based version of PowerShell on computers like the Microsoft Surface Pro X. For best results, install PowerShell to the `$Env:ProgramFiles\PowerShell\7` folder.

Install as a .NET Global tool

If you already have the .NET Core SDK installed, you can install PowerShell as a .NET Global tool.

```
dotnet tool install --global PowerShell
```

The dotnet tool installer adds `$HOME\.dotnet\tools` to your `$Env:PATH` environment variable. However, the currently running shell doesn't have the updated `$Env:PATH`. You can start PowerShell from a new shell by typing `pwsh`.

Installing from the Microsoft Store

PowerShell can be installed from the Microsoft Store. You can find the PowerShell release in the [Microsoft Store](#) site or in the Store application in Windows.

Benefits of the Microsoft Store package:

- Automatic updates built right into Windows

- Integrates with other software distribution mechanisms like Intune and Configuration Manager
- Can install on Windows systems using x86, x64, or Arm64 processors

Known limitations

By default, Windows Store packages run in an application sandbox that virtualizes access to some filesystem and registry locations. Changes to virtualized file and registry locations don't persist outside of the application sandbox.

This sandbox blocks all changes to the application's root folder. Any system-level configuration settings stored in `$PSHOME` can't be modified. This includes the WSMAN configuration. This prevents remote sessions from connecting to Store-based installs of PowerShell. User-level configurations and SSH remoting are supported.

The following commands need write to `$PSHOME`. These commands aren't supported in a Microsoft Store instance of PowerShell.

- `Register-PSSessionConfiguration`
- `Update-Help -Scope AllUsers`
- `Enable-ExperimentalFeature -Scope AllUsers`
- `Set-ExecutionPolicy -Scope LocalMachine`

For more information, see [Understanding how packaged desktop apps run on Windows](#).

Beginning in PowerShell 7.2, the PowerShell package is now exempt from file and registry virtualization. Changes to virtualized file and registry locations now persist outside of the application sandbox. However, changes to the application's root folder are still blocked.

 **Important**

You must be running on Windows build 1903 or higher for this exemption to work.

Installing a preview version

Preview releases of PowerShell 7 install to `$Env:ProgramFiles\PowerShell\7-preview` so they can be run side-by-side with non-preview releases of PowerShell. PowerShell 7.4 is the next preview release.

Upgrading an existing installation

For best results when upgrading, you should use the same install method you used when you first installed PowerShell. If you aren't sure how PowerShell was installed, you can check the value of the `$PSHOME` variable, which always points to the directory containing PowerShell that the current session is running.

- If the value is `$HOME\.dotnet\tools`, PowerShell was installed with the [.NET Global tool](#).
- If the value is `$Env:ProgramFiles\PowerShell\7`, PowerShell was installed as an [MSI package](#) or with [WinGet](#) on a computer with an X86 or x64 processor.
- If the value starts with `$Env:ProgramFiles\WindowsApps\`, PowerShell was installed as a [Microsoft Store package](#) or with [WinGet](#) on computer with an ARM processor.
- If the value is anything else, it's likely that PowerShell was installed as a [ZIP package](#).

If you installed via the MSI package, that information also appears in the [Programs and Features Control Panel](#).

To determine whether PowerShell may be upgraded with WinGet, run the following command:

```
PowerShell  
winget list --id Microsoft.PowerShell --upgrade-available
```

If there is an available upgrade, the output indicates the latest available version. Use the following command to upgrade PowerShell using WinGet:

```
PowerShell  
winget upgrade --id Microsoft.PowerShell
```

Deploying on Windows 10 IoT Enterprise

Windows 10 IoT Enterprise comes with Windows PowerShell, which we can use to deploy PowerShell 7.

```
PowerShell  
  
# Replace the placeholder information for the following variables:  
$deviceip = '<device ip address'  
$zipfile = 'PowerShell-7.5.1-win-arm64.zip'  
$downloadfolder = 'U:\Users\Administrator\Downloads' # The download location is local to the device.  
# There should be enough space for the zip file and the unzipped contents.  
  
# Create PowerShell session to target device  
Set-Item -Path WSMan:\localhost\Client\TrustedHosts $deviceip
```

```

$S = New-PSSession -ComputerName $deviceIp -Credential Administrator
# Copy the ZIP package to the device
Copy-Item $zipfile -Destination $downloadfolder -ToSession $S

#Connect to the device and expand the archive
Enter-PSSession $S
Set-Location U:\Users\Administrator\Downloads
Expand-Archive .\PowerShell-7.5.1-win-arm64.zip

# Set up remoting to PowerShell 7
Set-Location .\PowerShell-7.5.1-win-arm64
# Be sure to use the -PowerShellHome parameter otherwise it tries to create a new
# endpoint with Windows PowerShell 5.1
.\Install-PowerShellRemoting.ps1 -PowerShellHome .

```

When you set up PowerShell Remoting you get an error message and are disconnected from the device. PowerShell has to restart WinRM. Now you can connect to PowerShell 7 endpoint on device.

PowerShell

```

# Be sure to use the -Configuration parameter. If you omit it, you connect to
Windows PowerShell 5.1
Enter-PSSession -ComputerName $deviceIp -Credential Administrator -Configuration
PowerShell.7.5.1

```

Deploying on Windows 10 IoT Core

Windows 10 IoT Core adds Windows PowerShell when you include *IOT_POWERSHELL* feature, which we can use to deploy PowerShell 7. The steps defined above for Windows 10 IoT Enterprise can be followed for IoT Core as well.

For adding the latest PowerShell in the shipping image, use [Import-PSCoreRelease](#) command to include the package in the workarea and add *OPENSRC_POWERSHELL* feature to your image.

(!) Note

For ARM64 architecture, Windows PowerShell isn't added when you include *IOT_POWERSHELL*. So the zip based install doesn't work. You need to use [Import-PSCoreRelease](#) command to add it in the image.

Deploying on Nano Server

These instructions assume that the Nano Server is a "headless" OS that has a version of PowerShell already running on it. For more information, see the [Nano Server Image Builder](#) documentation.

PowerShell binaries can be deployed using two different methods.

1. Offline - Mount the Nano Server VHD and unzip the contents of the zip file to your chosen location within the mounted image.
2. Online - Transfer the zip file over a PowerShell Session and unzip it in your chosen location.

In both cases, you need the [Windows x64 ZIP release package](#). Run the commands within an "Administrator" instance of PowerShell.

Offline Deployment of PowerShell

1. Use your favorite zip utility to unzip the package to a directory within the mounted Nano Server image.
2. Unmount the image and boot it.
3. Connect to the built-in instance of Windows PowerShell.

Online Deployment of PowerShell

Deploy PowerShell to Nano Server using the following steps.

PowerShell

```
# Replace the placeholder information for the following variables:  
$ipaddr = '<Nano Server IP address>'  
$credential = Get-Credential # <An Administrator account on the system>  
$zipfile = 'PowerShell-7.5.1-win-x64.zip'  
# Connect to the built-in instance of Windows PowerShell  
$session = New-PSSession -ComputerName $ipaddr -Credential $credential  
# Copy the file to the Nano Server instance  
Copy-Item $zipfile C:\ -ToSession $session  
# Enter the interactive remote session  
Enter-PSSession $session  
# Extract the ZIP file  
Expand-Archive -Path C:\PowerShell-7.5.1-win-x64.zip -DestinationPath 'C:\Program  
Files\PowerShell 7'
```

PowerShell remoting

PowerShell supports the PowerShell Remoting Protocol (PSRP) over both WSMAN and SSH. For more information, see:

- [SSH Remoting in PowerShell](#)
- [WSMAN Remoting in PowerShell](#)

Supported versions of Windows

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [Windows reaches end-of-support](#).

- Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 for Windows Server 2022, Windows Server Core 2022, and Windows Server Nano build 1809 are available from the [Microsoft Artifact Registry](#).
- PowerShell 7.4 and higher can be installed on Windows 10 build 1607 and higher, Windows 11, Windows Server 2016 and higher.

Note

Support for a specific version of Windows is determined by the Microsoft Support Lifecycle policies. For more information, see:

- [Windows client lifecycle FAQ](#)
- [Modern Lifecycle Policy FAQ](#)

You can check the version that you are using by running `winver.exe`.

Installation support

Microsoft supports the installation methods in this document. There may be other third-party methods of installation available from other sources. While those tools and methods may work, Microsoft can't support those methods.

Install PowerShell on Linux

Article • 09/05/2023

PowerShell can be installed on several different Linux distributions. Most Linux platforms and distributions have a major release each year, and provide a package manager that's used to install PowerShell. PowerShell can be installed on some distributions of Linux that aren't supported by Microsoft. In those cases, you may find support from the community for PowerShell on those platforms.

For more information, see the [PowerShell Support Lifecycle](#) documentation.

This article lists the supported Linux distributions and package managers. All PowerShell releases remain supported until either the version of PowerShell or the version of the Linux distribution reaches end-of-support.

For the best compatibility, choose a long-term release (LTS) version.

Alpine

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [Alpine reaches end-of-life](#).

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 are available from the [Microsoft Artifact Registry](#) for the following versions of Alpine:

- Alpine 3.20 - OS support ends on 2026-04-01

Docker images of PowerShell aren't available for Alpine 3.21.

Important

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates. Microsoft recommends that you update the OS packages to the latest version to ensure the latest security updates are applied.

For more information, see [Install PowerShell on Alpine](#).

Debian

Debian uses APT (Advanced Package Tool) as a package manager.

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [Debian reaches end-of-life](#).

Install package files (.deb) are also available from <https://packages.microsoft.com/>.

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 are available from the [Microsoft Artifact Registry](#) for the following versions of Debian:

- Debian 12 (Bookworm) - OS support ends on 2026-06-10

Important

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates. Microsoft recommends that you update the OS packages to the latest version to ensure the latest security updates are applied.

For more information, see [Install PowerShell on Debian](#).

Red Hat Enterprise Linux (RHEL)

RHEL 7 uses yum and RHEL 8 uses the dnf package manager.

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [RHEL reaches end-of-support](#).

Install package files (.rpm) are also available from <https://packages.microsoft.com/>.

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 are available from the [Microsoft Artifact Registry](#) for the following versions of RHEL:

- RHEL 9 - OS support ends on 2032-05-31
- RHEL 8 - OS support ends on 2029-05-31

PowerShell is tested on Red Hat Universal Base Images (UBI). For more information, see the [UBI information page](#).

Important

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates. Microsoft recommends

that you update the OS packages to the latest version to ensure the latest security updates are applied.

For more information, see [Install PowerShell on RHEL](#).

Ubuntu

Ubuntu uses APT (Advanced Package Tool) as a package manager.

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [Ubuntu reaches end-of-support](#).

Install package files (.deb) are also available from <https://packages.microsoft.com/>.

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 and Arm32 are available from the [Microsoft Artifact Registry](#) for the following versions of Ubuntu:

- Ubuntu 24.04 (Noble Numbat) - OS support ends on 2029-04-01
- Ubuntu 22.04 (Jammy Jellyfish) - OS support ends on 2027-04-01
- Ubuntu 20.04 (Focal Fossa) - OS support ends on 2025-04-02

Ubuntu 24.10 (Oracular Oriole) is an interim release. Microsoft doesn't support [interim releases](#) of Ubuntu. For more information, see [Community supported distributions](#).

Important

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates. Microsoft recommends that you update the OS packages to the latest version to ensure the latest security updates are applied.

For more information, see [Install PowerShell on Ubuntu](#).

Community supported distributions

PowerShell can be installed on many distributions of Linux that aren't supported by Microsoft. In those cases, you may find support from the community for PowerShell on those platforms.

To be supported by Microsoft, the Linux distribution must meet the following criteria:

- The version and architecture of the distribution is supported by .NET Core.
- The version of the distribution is supported for at least one year.

- The version of the distribution isn't an interim release or equivalent.
- The PowerShell team has tested the version of the distribution.

For more information, see [Community support for PowerShell on Linux](#).

Alternate installation methods

There are three other ways to install PowerShell on Linux, including Linux distributions that aren't officially supported. You can try to install PowerShell using the PowerShell Snap Package. You can also try deploying PowerShell binaries directly using the Linux `tar.gz` package. For more information, see [Alternate ways to install PowerShell on Linux](#).

Installing PowerShell on Alpine Linux

Article • 04/28/2025

All packages are available on our GitHub [releases](#) page. After the package is installed, run `pwsh` from a terminal. Run `pwsh-preview` if you installed a preview release. Before installing, check the list of [Supported versions](#) below.

ⓘ Note

PowerShell 7.4 is an in-place upgrade that removes previous versions of PowerShell 7. Preview versions of PowerShell can be installed side-by-side with other versions of PowerShell. If you need to run PowerShell 7.4 side-by-side with a previous version, reinstall the previous version using the [binary archive](#) method.

ⓘ Note

The installation commands in this article are for the latest stable release of PowerShell. To install a different version of PowerShell, adjust the command to match the version you need. The following links direct you to the release page for each version in the PowerShell repository on GitHub.

- v7.5.0 - Stable release: <https://aka.ms/powershell-release?tag=stable>
- v7.4.7 - LTS release: <https://aka.ms/powershell-release?tag=lts>
- v7.6.0-preview.2 - Preview release: <https://aka.ms/powershell-release?tag=preview>

Download links for every package are found in the **Assets** section of the Release page. The **Assets** section may be collapsed, so you may need to click to expand it.

Installation steps

Installation on Alpine is based on downloading tar.gz package from the [releases](#) page. The URL to the package depends on the version of PowerShell you want to install.

- PowerShell 7.4 -
<https://github.com/PowerShell/PowerShell/releases/download/v7.4.7/powershell-7.4.7-linux-musl-x64.tar.gz>
- PowerShell 7.5 -
<https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell-7.5.1-linux-musl-x64.tar.gz>

Then, in the terminal, execute the following shell commands to install PowerShell 7.4:

```
sh

# install the requirements
sudo apk add --no-cache \
    ca-certificates \
    less \
    ncurses-terminfo-base \
    krb5-libs \
    libgcc \
    libintl \
    libssl3 \
    libstdc++ \
    tzdata \
    userspace-rcu \
    zlib \
    icu-libs \
    curl

apk -X https://dl-cdn.alpinelinux.org/alpine/edge/main add --no-cache \
    lttng-ust \
    openssh-client \

# Download the powershell '.tar.gz' archive
curl -L
https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell-
7.5.1-linux-musl-x64.tar.gz -o /tmp/powershell.tar.gz

# Create the target folder where powershell will be placed
sudo mkdir -p /opt/microsoft/powershell/7

# Expand powershell to the target folder
sudo tar zxf /tmp/powershell.tar.gz -C /opt/microsoft/powershell/7

# Set execute permissions
sudo chmod +x /opt/microsoft/powershell/7/pwsh

# Create the symbolic link that points to pwsh
sudo ln -s /opt/microsoft/powershell/7/pwsh /usr/bin/pwsh

# Start PowerShell
pwsh
```

Uninstall PowerShell

```
sh

sudo rm -rf /usr/bin/pwsh /opt/microsoft/powershell
```

PowerShell paths

- `$PSHOME` is `/opt/microsoft/powershell/7/`
- The profiles scripts are stored in the following locations:
 - AllUsersAllHosts - `$PSHOME/profile.ps1`
 - AllUsersCurrentHost - `$PSHOME/Microsoft.PowerShell_profile.ps1`
 - CurrentUserAllHosts - `~/.config/powershell/profile.ps1`
 - CurrentUserCurrentHost - `~/.config/powershell/Microsoft.PowerShell_profile.ps1`
- Modules are stored in the following locations:
 - User modules - `~/.local/share/powershell/Modules`
 - Shared modules - `/usr/local/share/powershell/Modules`
 - Default modules - `$PSHOME/Modules`
- PSReadLine history is recorded in
`~/.local/share/powershell/PSReadLine/ConsoleHost_history.txt`

The profiles respect PowerShell's per-host configuration, so the default host-specific profiles exists at `Microsoft.PowerShell_profile.ps1` in the same locations.

PowerShell respects the [XDG Base Directory Specification](#) on Linux.

Supported versions

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [Alpine reaches end-of-life](#).

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 are available from the [Microsoft Artifact Registry](#) for the following versions of Alpine:

- Alpine 3.20 - OS support ends on 2026-04-01

Docker images of PowerShell aren't available for Alpine 3.21.

Important

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates. Microsoft recommends that you update the OS packages to the latest version to ensure the latest security updates are applied.

Installation support

Microsoft supports the installation methods in this document. There may be other methods of installation available from other third-party sources. While those tools and methods may work, Microsoft can't support those methods.

Installing PowerShell on Debian

Article • 04/28/2025

All packages are available on our GitHub [releases](#) page. Before installing, check the list of [Supported versions](#) below. After the package is installed, run `pwsh` from a terminal. Run `pwsh-lts` if you installed a preview release.

⚠ Note

PowerShell 7.4 is an in-place upgrade that removes previous versions of PowerShell 7. Preview versions of PowerShell can be installed side-by-side with other versions of PowerShell. If you need to run PowerShell 7.4 side-by-side with a previous version, reinstall the previous version using the [binary archive](#) method.

Debian uses APT (Advanced Package Tool) as a package manager.

⚠ Note

The installation commands in this article are for the latest stable release of PowerShell. To install a different version of PowerShell, adjust the command to match the version you need. The following links direct you to the release page for each version in the PowerShell repository on GitHub.

- v7.5.0 - Stable release: <https://aka.ms/powershell-release?tag=stable>
- v7.4.7 - LTS release: <https://aka.ms/powershell-release?tag=lts>
- v7.6.0-preview.2 - Preview release: <https://aka.ms/powershell-release?tag=preview>

Download links for every package are found in the **Assets** section of the Release page. The **Assets** section may be collapsed, so you may need to click to expand it.

Installation on Debian 11 or 12 via the Package Repository

Microsoft builds and supports a variety of software products for Linux systems and makes them available via Linux packaging clients (apt, dnf, yum, etc). These Linux software packages are hosted on the *Linux package repository for Microsoft products*, <https://packages.microsoft.com>, also known as *PMC*.

Installing PowerShell from PMC is the preferred method of installation.

⚠ Note

This script only works for supported versions of Debian.

```
sh

#####
# Prerequisites

# Update the list of packages
sudo apt-get update

# Install pre-requisite packages.
sudo apt-get install -y wget

# Get the version of Debian
source /etc/os-release

# Download the Microsoft repository GPG keys
wget -q https://packages.microsoft.com/config/debian/$VERSION_ID/packages-microsoft-prod.deb

# Register the Microsoft repository GPG keys
sudo dpkg -i packages-microsoft-prod.deb

# Delete the Microsoft repository GPG keys file
rm packages-microsoft-prod.deb

# Update the list of packages after we added packages.microsoft.com
sudo apt-get update

#####
# Install PowerShell
sudo apt-get install -y powershell

# Start PowerShell
pwsh
```

Installation via direct download

PowerShell 7.2 introduced a universal package that makes installation easier. Download the universal package from the [releases ↗](#) page onto your Debian machine.

The link to the current version is:

- PowerShell 7.4 (LTS) universal package for supported versions of Debian
 - https://github.com/PowerShell/PowerShell/releases/download/v7.4.7/powershell_7.4.7-1.deb_amd64.deb

- PowerShell 7.5 universal package for supported versions of Debian
 - https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell_7.5.1-1.deb_amd64.deb

The following shell script downloads and installs the current release of PowerShell. You can change the URL to download the version of PowerShell that you want to install.

```
sh

#####
# Prerequisites

# Update the list of packages
sudo apt-get update

# Install pre-requisite packages.
sudo apt-get install -y wget

# Download the PowerShell package file
wget
https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell_7.5.1-1.deb_amd64.deb

#####
# Install the PowerShell package
sudo dpkg -i powershell_7.5.1-1.deb_amd64.deb

# Resolve missing dependencies and finish the install (if necessary)
sudo apt-get install -f

# Delete the downloaded package file
rm powershell_7.5.1-1.deb_amd64.deb

# Start PowerShell
pwsh
```

Uninstall PowerShell

```
sh

sudo apt-get remove powershell
```

PowerShell paths

- \$PSHOME is /opt/microsoft/powershell/7/
- The profiles scripts are stored in the following locations:

- AllUsersAllHosts - `$PSHOME/profile.ps1`
- AllUsersCurrentHost - `$PSHOME/Microsoft.PowerShell_profile.ps1`
- CurrentUserAllHosts - `~/.config/powershell/profile.ps1`
- CurrentUserCurrentHost - `~/.config/powershell/Microsoft.PowerShell_profile.ps1`
- Modules are stored in the following locations:
 - User modules - `~/.local/share/powershell/Modules`
 - Shared modules - `/usr/local/share/powershell/Modules`
 - Default modules - `$PSHOME/Modules`
- PSReadLine history is recorded in
`~/.local/share/powershell/PSReadLine/ConsoleHost_history.txt`

PowerShell respects the [XDG Base Directory Specification](#) on Linux.

Supported versions

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [Debian reaches end-of-life](#).

Install package files (`.deb`) are also available from <https://packages.microsoft.com/>.

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 are available from the [Microsoft Artifact Registry](#) for the following versions of Debian:

- Debian 12 (Bookworm) - OS support ends on 2026-06-10

Important

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates. Microsoft recommends that you update the OS packages to the latest version to ensure the latest security updates are applied.

Installation support

Microsoft supports the installation methods in this document. There may be other methods of installation available from other third-party sources. While those tools and methods may work, Microsoft can't support those methods.

Installing PowerShell on Red Hat Enterprise Linux (RHEL)

Article • 04/28/2025

All packages are available on our GitHub [releases](#) page. Before installing, check the list of [Supported versions](#) below. After the package is installed, run `pwsh` from a terminal. Run `pwsh-preview` if you installed a preview release.

⚠ Note

PowerShell 7.4 is an in-place upgrade that removes previous versions of PowerShell 7. Preview versions of PowerShell can be installed side-by-side with other versions of PowerShell. If you need to run PowerShell 7.4 side-by-side with a previous version, reinstall the previous version using the [binary archive](#) method.

RHEL 7 uses `yum` and RHEL 8 and higher uses the `dnf` package manager.

⚠ Note

The installation commands in this article are for the latest stable release of PowerShell. To install a different version of PowerShell, adjust the command to match the version you need. The following links direct you to the release page for each version in the PowerShell repository on GitHub.

- v7.5.0 - Stable release: <https://aka.ms/powershell-release?tag=stable>
- v7.4.7 - LTS release: <https://aka.ms/powershell-release?tag=lts>
- v7.6.0-preview.2 - Preview release: <https://aka.ms/powershell-release?tag=preview>

Download links for every package are found in the **Assets** section of the Release page. The **Assets** section may be collapsed, so you may need to click to expand it.

Installation via the Package Repository

Microsoft builds and supports a variety of software products for Linux systems and makes them available via Linux packaging clients (apt, dnf, yum, etc). These Linux software packages are hosted on the *Linux package repository for Microsoft products*, <https://packages.microsoft.com>, also known as *PMC*.

Installing PowerShell from PMC is the preferred method of installation.

ⓘ Note

This script only works for supported versions of RHEL.

```
sh

#####
# Prerequisites

# Get version of RHEL
source /etc/os-release
if [ ${VERSION_ID%.} -lt 8 ]
then majorver=7
elif [ ${VERSION_ID%.} -lt 9 ]
then majorver=8
else majorver=9
fi

# Download the Microsoft RedHat repository package
curl -sSL -O https://packages.microsoft.com/config/rhel/$majorver/packages-
microsoft-prod.rpm

# Register the Microsoft RedHat repository
sudo rpm -i packages-microsoft-prod.rpm

# Delete the downloaded package after installing
rm packages-microsoft-prod.rpm

# Update package index files
sudo dnf update
# Install PowerShell
sudo dnf install powershell -y
```

Installation via direct download

PowerShell 7.2 introduced a universal package that makes installation easier. Download the universal package from the [releases ↗](#) page onto your RHEL machine.

The link to the current version is:

- PowerShell 7.4.7 universal package for supported versions of RHEL
 - https://github.com/PowerShell/PowerShell/releases/download/v7.4.7/powershell-7.4.7-1.rh.x86_64.rpm
- PowerShell 7.5.1 universal package for supported versions of RHEL
 - https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell-7.5.1-1.rh.x86_64.rpm

The following shell script downloads and installs the current preview release of PowerShell. You can change the URL to download the version of PowerShell that you want to install.

On RHEL 8 or 9:

```
sh  
  
sudo dnf install  
https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell-  
7.5.1-1.rh.x86_64.rpm
```

Uninstall PowerShell

On RHEL 8 or 9:

```
sh  
  
sudo dnf remove powershell
```

Support for Arm processors

PowerShell 7.2 and newer supports running on RHEL using a 64-bit Arm processor. Use the binary archive installation method of installing PowerShell that's described in [Alternate ways to install PowerShell on Linux](#).

PowerShell paths

- `$PSHOME` is `/opt/microsoft/powershell/7/`
- The profiles scripts are stored in the following locations:
 - AllUsersAllHosts - `$PSHOME/profile.ps1`
 - AllUsersCurrentHost - `$PSHOME/Microsoft.PowerShell_profile.ps1`
 - CurrentUserAllHosts - `~/.config/powershell/profile.ps1`
 - CurrentUserCurrentHost - `~/.config/powershell/Microsoft.PowerShell_profile.ps1`
- Modules are stored in the following locations:
 - User modules - `~/.local/share/powershell/Modules`
 - Shared modules - `/usr/local/share/powershell/Modules`
 - Default modules - `$PSHOME/Modules`
- PSReadLine history is recorded in
`~/.local/share/powershell/PSReadLine/ConsoleHost_history.txt`

PowerShell respects the [XDG Base Directory Specification](#) on Linux.

Supported versions

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [RHEL reaches end-of-support](#).

Install package files (`.rpm`) are also available from <https://packages.microsoft.com/>.

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 are available from the [Microsoft Artifact Registry](#) for the following versions of RHEL:

- RHEL 9 - OS support ends on 2032-05-31
- RHEL 8 - OS support ends on 2029-05-31

PowerShell is tested on Red Hat Universal Base Images (UBI). For more information, see the [UBI information page](#).

Important

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates. Microsoft recommends that you update the OS packages to the latest version to ensure the latest security updates are applied.

Installation support

Microsoft supports the installation methods in this document. There may be other methods of installation available from other third-party sources. While those tools and methods may work, Microsoft can't support those methods.

Installing PowerShell on Ubuntu

Article • 04/28/2025

All packages are available on our GitHub [releases](#) page. Before installing, check the list of [Supported versions](#) below. After the package is installed, run `pwsh` from a terminal. Run `pwsh-lts` if you installed a preview release.

ⓘ Note

PowerShell 7.4 is an in-place upgrade that removes previous versions of PowerShell 7. Preview versions of PowerShell can be installed side-by-side with other versions of PowerShell. If you need to run PowerShell 7.4 side-by-side with a previous version, reinstall the previous version using the [binary archive](#) method.

Ubuntu uses APT (Advanced Package Tool) as a package manager.

ⓘ Note

The installation commands in this article are for the latest stable release of PowerShell. To install a different version of PowerShell, adjust the command to match the version you need. The following links direct you to the release page for each version in the PowerShell repository on GitHub.

- v7.5.0 - Stable release: <https://aka.ms/powershell-release?tag=stable>
- v7.4.7 - LTS release: <https://aka.ms/powershell-release?tag=lts>
- v7.6.0-preview.2 - Preview release: <https://aka.ms/powershell-release?tag=preview>

Download links for every package are found in the **Assets** section of the Release page. The **Assets** section may be collapsed, so you may need to click to expand it.

Installation via Package Repository the Package Repository

Microsoft builds and supports a variety of software products for Linux systems and makes them available via Linux packaging clients (apt, dnf, yum, etc). These Linux software packages are hosted on the *Linux package repository for Microsoft products*, <https://packages.microsoft.com>, also known as *PMC*.

Installing PowerShell from PMC is the preferred method of installation.

ⓘ Note

This script only works for supported versions of Ubuntu.

```
sh

#####
# Prerequisites

# Update the list of packages
sudo apt-get update

# Install pre-requisite packages.
sudo apt-get install -y wget apt-transport-https software-properties-common

# Get the version of Ubuntu
source /etc/os-release

# Download the Microsoft repository keys
wget -q https://packages.microsoft.com/config/ubuntu/$VERSION_ID/packages-microsoft-prod.deb

# Register the Microsoft repository keys
sudo dpkg -i packages-microsoft-prod.deb

# Delete the Microsoft repository keys file
rm packages-microsoft-prod.deb

# Update the list of packages after we added packages.microsoft.com
sudo apt-get update

#####
# Install PowerShell
sudo apt-get install -y powershell

# Start PowerShell
pwsh
```

ⓘ Important

Ubuntu comes preconfigured with a package repository that includes .NET packages, but not PowerShell. Using these instructions to install PowerShell registers the Microsoft repository as a package source. You can install PowerShell and some versions of .NET from this repository. However, the Ubuntu package repository has different versions of the .NET packages. This can cause problems when installing .NET for other purposes. For more information about these problems, see [Troubleshoot .NET package mix ups on Linux](#).

You must choose the feed you want to use to install .NET. You can set the priority of the package repositories to favor one over the other. For instructions on how to set the priorities, see [My Linux distribution provides .NET packages, and I want to use them](#).

Installation via direct download

PowerShell 7.2 introduced a universal package that makes installation easier. Download the universal package from the [releases ↗](#) page onto your Ubuntu machine.

The link to the current version is:

- PowerShell 7.4 (LTS) universal package for supported versions of Ubuntu
 - https://github.com/PowerShell/PowerShell/releases/download/v7.4.7/powershell_7.4.7-1.deb_amd64.deb
- PowerShell 7.5 universal package for supported versions of Ubuntu
 - https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell-preview_7.5.1-1.deb_amd64.deb

The following shell script downloads and installs the current preview release of PowerShell. You can change the URL to download the version of PowerShell that you want to install.

```
sh

#####
# Prerequisites

# Update the list of packages
sudo apt-get update

# Install pre-requisite packages.
sudo apt-get install -y wget

# Download the PowerShell package file
wget
https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell_7.5.1-1.deb_amd64.deb

#####
# Install the PowerShell package
sudo dpkg -i powershell_7.5.1-1.deb_amd64.deb

# Resolve missing dependencies and finish the install (if necessary)
sudo apt-get install -f

# Delete the downloaded package file
rm powershell_7.5.1-1.deb_amd64.deb
```

```
# Start PowerShell Preview  
pwsh
```

Uninstall PowerShell

```
sh  
  
sudo apt-get remove powershell
```

Support for Arm processors

PowerShell 7.2 and newer supports running on Ubuntu using 32-bit Arm processors. Use the binary archive installation method of installing PowerShell that's described in [Alternate ways to install PowerShell on Linux](#).

PowerShell paths

- `$PSHOME` is `/opt/microsoft/powershell/7/`
- The profiles scripts are stored in the following locations:
 - AllUsersAllHosts - `$PSHOME/profile.ps1`
 - AllUsersCurrentHost - `$PSHOME/Microsoft.PowerShell_profile.ps1`
 - CurrentUserAllHosts - `~/.config/powershell/profile.ps1`
 - CurrentUserCurrentHost - `~/.config/powershell/Microsoft.PowerShell_profile.ps1`
- Modules are stored in the following locations:
 - User modules - `~/.local/share/powershell/Modules`
 - Shared modules - `/usr/local/share/powershell/Modules`
 - Default modules - `$PSHOME/Modules`
- PSReadLine history is recorded in
`~/.local/share/powershell/PSReadLine/ConsoleHost_history.txt`

PowerShell respects the [XDG Base Directory Specification](#) on Linux.

Supported versions

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [Ubuntu reaches end-of-support](#).

Install package files (`.deb`) are also available from <https://packages.microsoft.com/>.

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 and Arm32 are available from the [Microsoft Artifact Registry](#) for the following versions of Ubuntu:

- Ubuntu 24.04 (Noble Numbat) - OS support ends on 2029-04-01
- Ubuntu 22.04 (Jammy Jellyfish) - OS support ends on 2027-04-01
- Ubuntu 20.04 (Focal Fossa) - OS support ends on 2025-05-31

Ubuntu 24.10 (Oracular Oriole) is an interim release. Microsoft doesn't support [interim releases](#) of Ubuntu. For more information, see [Community supported distributions](#).

 **Important**

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates. Microsoft recommends that you update the OS packages to the latest version to ensure the latest security updates are applied.

Installation support

Microsoft supports the installation methods in this document. There may be other methods of installation available from other third-party sources. While those tools and methods may work, Microsoft can't support those methods.

Community support for PowerShell on Linux

Article • 05/13/2025

You can install PowerShell on some distributions of Linux that aren't supported by Microsoft. In those cases, you might find support from the community for PowerShell on those platforms.

Supported Linux distributions must meet the following criteria:

- The version and architecture of the distribution is supported by .NET Core.
- The version of the distribution is supported for at least one year.
- The version of the distribution isn't an interim release or equivalent.
- The PowerShell team has tested the version of the distribution.

For more information, see the [PowerShell Support Lifecycle](#) documentation.

The following distributions are examples of distributions supported by the community. Each distribution has its own community support mechanisms. Consult the distribution's website to find their community resources. You can also get help from these [PowerShell Community](#) resources.

Ubuntu interim releases

The documented steps to install PowerShell on [Ubuntu](#) might work on Ubuntu interim releases. However, Microsoft only supports PowerShell on the Long Term Servicing (LTS) releases of Ubuntu. Microsoft doesn't support [interim releases](#) of Ubuntu.

Arch Linux

PowerShell is available from the [Arch Linux](#) User Repository (AUR). Packages in the AUR are maintained by the Arch community. To install the [latest release binary](#), see the [Arch Linux wiki](#) or [Using PowerShell in Docker](#).

Kali

Installation - Kali

```
sh
```

```
# Install PowerShell package
apt update && apt -y install powershell

# Start PowerShell
pwsh
```

Uninstallation - Kali

```
sh

# Uninstall PowerShell package
apt -y remove powershell
```

Gentoo

You can install PowerShell on Gentoo Linux using packages from the Gentoo package repository. For information about installing these packages, see the [PowerShell](#) page in the Gentoo wiki.

SLES and openSUSE

You may be able to install PowerShell on SLES and openSUSE using the SNAP package manager. Also, the following article provides information on how to install PowerShell on openSUSE:

- [PowerShell - openSUSE Wiki](#)

Raspberry Pi OS

[Raspberry Pi OS](#) (formerly Raspbian) is a free operating system based on Debian.

Important

.NET isn't supported on ARMv6 architecture devices, including Raspberry Pi Zero and Raspberry Pi devices released before Raspberry Pi 2.

Install on Raspberry Pi OS

Download the tar.gz package from the [releases](#) page onto your Raspberry Pi computer. The links to the current versions are:

- PowerShell 7.4 - latest LTS release
 - <https://github.com/PowerShell/PowerShell/releases/download/v7.4.10/powershell-7.4.10-linux-arm32.tar.gz>
 - <https://github.com/PowerShell/PowerShell/releases/download/v7.4.10/powershell-7.4.10-linux-arm64.tar.gz>
- PowerShell 7.5 - latest stable release
 - <https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell-7.5.1-linux-arm32.tar.gz>
 - <https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell-7.5.1-linux-arm64.tar.gz>

Use the following shell commands to download and install the package. This script detects whether you're running a 32-bit or 64-bit OS and installs the latest stable version of PowerShell for that processor type.

```
sh

#####
# Prerequisites

# Update package lists
sudo apt-get update

# Install dependencies
sudo apt-get install jq libssl1.1 libunwind8 -y

#####
# Download and extract PowerShell

# Grab the latest tar.gz
bits=$(getconf LONG_BIT)
release=$(curl -sL
https://api.github.com/repos/PowerShell/PowerShell/releases/latest)
package=$(echo $release | jq -r ".assets[].browser_download_url" | grep "linux-
arm${bits}.tar.gz")
wget $package

# Make folder to put powershell
mkdir ~/powershell

# Unpack the tar.gz file
tar -xvf "./${package##*/}" -C ~/powershell

# Start PowerShell
~/powershell/pwsh
```

Optionally, you can create a symbolic link to start PowerShell without specifying the path to the `pwsh` binary.

```
sh

# Start PowerShell from bash with sudo to create a symbolic link
sudo ~/powershell/pwsh -Command 'New-Item -ItemType SymbolicLink -Path
"/usr/bin/pwsh" -Target "$PSHOME/pwsh" -Force'

# alternatively you can run following to create a symbolic link
# sudo ln -s ~/powershell/pwsh /usr/bin/pwsh

# Now to start PowerShell you can just run "pwsh"
```

Uninstallation - Raspberry Pi OS

```
sh

rm -rf ~/powershell
```

Alternate ways to install PowerShell on Linux

Article • 04/28/2025

All packages are available on our GitHub [releases](#) page. After the package is installed, run `pwsh` from a terminal. Run `pwsh-preview` if you installed a preview release.

There are three other ways to install PowerShell on a Linux distribution:

- Install using a [Snap Package](#)
- Install using the [binary archives](#)
- Install as a [.NET Global tool](#)

Snap Package

Snaps are application packages that are easy to install, secure, cross-platform and dependency-free. Snaps are discoverable and installable from the Snap Store. Snap packages are supported the same as the distribution you're running the package on.

Important

The Snap Store contains PowerShell snap packages for many Linux distributions that are not officially supported by Microsoft. For support, see the list of available [Community Support](#) options.

Getting snapd

`snapd` is required to run snaps. Use [these instructions](#) to make sure you have `snapd` installed.

Installation via Snap

There are two PowerShell for Linux is published to the [Snap store](#): `powershell` and `powershell-preview`.

Use the following command to install the latest stable version of PowerShell:

```
sh  
  
# Install PowerShell  
sudo snap install powershell --classic
```

```
# Start PowerShell  
pwsh
```

If you don't specify the `--channel` parameter, Snap installs the latest stable version. To install the latest LTS version, use the following method:

```
sh  
  
# Install PowerShell  
sudo snap install powershell --channel=lts/stable --classic  
  
# Start PowerShell  
pwsh
```

ⓘ Note

Microsoft only supports the `latest/stable` and `lts/stable` channels for the `powershell` package. Do not install packages from the other channels.

To install a preview version, use the following method:

```
sh  
  
# Install PowerShell  
sudo snap install powershell-preview --classic  
  
# Start PowerShell  
pwsh-preview
```

ⓘ Note

Microsoft only supports the `latest/stable` channel for the `powershell-preview` package. Do not install packages from the other channels.

After installation, Snap will automatically upgrade. You can trigger an upgrade using `sudo snap refresh powershell` or `sudo snap refresh powershell-preview`.

Uninstallation

```
sh  
  
sudo snap remove powershell
```

or

```
sh  
sudo snap remove powershell-preview
```

Binary Archives

PowerShell binary `tar.gz` archives are provided for Linux platforms to enable advanced deployment scenarios.

!*Note*

You can use this method to install any version of PowerShell including the latest:

- Stable release: <https://aka.ms/powershell-release?tag=stable>
- LTS release: <https://aka.ms/powershell-release?tag=lts>
- Preview release: <https://aka.ms/powershell-release?tag=preview>

Dependencies

PowerShell builds portable binaries for all Linux distributions. But, .NET Core runtime requires different dependencies on different distributions, and PowerShell does too.

It's possible that when you install PowerShell, specific dependencies may not be installed, such as when manually installing from the binary archives. The following list details Linux distributions that are supported by Microsoft and have dependencies you may need to install. Check the distribution page for more information:

- [Alpine](#)
- [Debian](#)
- [RHEL](#)
- [SLES](#)
- [Ubuntu](#)

To deploy PowerShell binaries on Linux distributions that aren't officially supported, you need to install the necessary dependencies for the target OS in separate steps. For example, our [Amazon Linux dockerfile](#) installs dependencies first, and then extracts the Linux `tar.gz` archive.

Installation using a binary archive file

ⓘ Important

This method can be used to install PowerShell on any version of Linux, including distributions that are not officially supported by Microsoft. Be sure to install any necessary dependencies. For support, see the list of available [Community Support](#) options.

The following example shows the steps for installing the x64 binary archive. You must choose the correct binary archive that matches the processor type for your platform.

- `powershell-7.5.1-linux-arm32.tar.gz`
- `powershell-7.5.1-linux-arm64.tar.gz`
- `powershell-7.5.1-linux-x64.tar.gz`

Use the following shell commands to download and install PowerShell from the `.tar.gz` binary archive. Change the URL to match the version of PowerShell you want to install.

```
sh

# Download the powershell '.tar.gz' archive
curl -L -o /tmp/powershell.tar.gz
https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell-
7.5.1-linux-x64.tar.gz

# Create the target folder where powershell will be placed
sudo mkdir -p /opt/microsoft/powershell/7

# Expand powershell to the target folder
sudo tar zxf /tmp/powershell.tar.gz -C /opt/microsoft/powershell/7

# Set execute permissions
sudo chmod +x /opt/microsoft/powershell/7/pwsh

# Create the symbolic link that points to pwsh
sudo ln -s /opt/microsoft/powershell/7/pwsh /usr/bin/pwsh
```

Uninstalling binary archives

```
sh

sudo rm -rf /usr/bin/pwsh /opt/microsoft/powershell
```

Install as a .NET Global tool

If you already have the [.NET Core SDK](#) installed, it's easy to install PowerShell as a [.NET Global tool](#).

```
sh  
dotnet tool install --global PowerShell
```

The dotnet tool installer adds `~/.dotnet/tools` to your `PATH` environment variable. However, the currently running shell does not have the updated `PATH`. You should be able to start PowerShell from a new shell by typing `pwsh`.

Installing PowerShell on macOS

Article • 04/29/2025

PowerShell 7 or higher requires macOS 11 and higher. All packages are available on our GitHub [releases](#) page. After the package is installed, run `pwsh` from a terminal. Before installing, check the list of [Supported versions](#) below.

ⓘ Note

PowerShell 7.4 is an in-place upgrade that removes previous versions of PowerShell 7.

Preview versions of PowerShell can be installed side-by-side with other versions of PowerShell. If you need to run PowerShell 7.4 side-by-side with a previous version, reinstall the previous version using the [binary archive](#) method.

ⓘ Note

The installation commands in this article are for the latest stable release of PowerShell. To install a different version of PowerShell, adjust the command to match the version you need. The following links direct you to the release page for each version in the PowerShell repository on GitHub.

- v7.5.0 - Stable release: <https://aka.ms/powershell-release?tag=stable>
- v7.4.7 - LTS release: <https://aka.ms/powershell-release?tag=lts>
- v7.6.0-preview.2 - Preview release: <https://aka.ms/powershell-release?tag=preview>

Download links for every package are found in the **Assets** section of the Release page. The **Assets** section may be collapsed, so you may need to click to expand it.

Install the latest stable release of PowerShell

There are several ways to install PowerShell on macOS. Choose one of the following methods:

- Install using [Homebrew](#). Homebrew is the preferred package manager for macOS.
- Install PowerShell via [Direct Download](#)
- Install from [binary archives](#).

If the `brew` command isn't found, you need to install Homebrew following [their instructions](#).

Bash

```
/bin/bash -c "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Once `brew` is installed you can install PowerShell.

The following command installs the latest stable release of PowerShell:

```
sh  
  
brew install --cask powershell
```

Finally, verify that your install is working properly:

```
sh  
  
pwsh
```

When new versions of PowerShell are released, update Homebrew's formulae and upgrade PowerShell:

```
sh  
  
brew update  
brew upgrade powershell
```

ⓘ Note

The commands above can be called from within a PowerShell (`pwsh`) host, but then the PowerShell shell must be exited and restarted to complete the upgrade and refresh the values shown in `$PSVersionTable`.

Install the latest preview release of PowerShell

After you've installed Homebrew, you can install PowerShell.

```
sh  
  
brew install powershell/tap/powershell-preview
```

Run the following command to start the preview version of PowerShell:

```
sh
```

```
pwsh-preview
```

When new versions of PowerShell are released, update Homebrew's formulae and upgrade PowerShell:

```
sh
```

```
brew update  
brew upgrade powershell-preview
```

 **Note**

The commands above can be called from within a PowerShell (pwsh) host, but then the PowerShell shell must be exited and restarted to complete the upgrade, and refresh the values shown in `$PSVersionTable`.

Install the latest LTS release of PowerShell

```
sh
```

```
brew install powershell/tap/powershell-lts
```

You can now verify your install

```
sh
```

```
pwsh-lts
```

When new versions of PowerShell are released, run the following command.

```
sh
```

```
brew upgrade powershell-lts
```

 **Note**

Whether you use the cask or the tap method, when updating to a newer version of PowerShell, use the same method you used to initially install PowerShell. If you use a

different method, opening a new pwsh session will continue to use the older version of PowerShell.

If you do decide to use different methods, there are ways to correct the issue using the [Homebrew link method](#).

Installation via Direct Download

Starting with version 7.2, PowerShell supports the Apple M-series Arm-based processors. Download the install package from the [releases](#) page onto your computer. The links to the current versions are:

- PowerShell 7.4
 - x64 processors - [powershell-7.4.7-osx-x64.pkg](#)
 - Arm64 processors - [powershell-7.4.7-osx-arm64.pkg](#)
- PowerShell 7.5
 - x64 processors - [powershell-7.5.1-osx-x64.pkg](#)
 - Arm64 processors - [powershell-7.5.1-arm64.pkg](#)

You can double-click the file and follow the prompts, or install it from the terminal using the following commands. Change the name of the file to match the file you downloaded.

```
sh  
sudo installer -pkg ./Downloads/powershell-7.5.1-osx-x64.pkg -target /
```

If you are running on macOS Big Sur 11.5 or higher you may receive the following error message when installing the package:

"powershell-7.5.1-osx-x64.pkg" cannot be opened because Apple cannot check it for malicious software.

There are two ways to work around this issue:

Using the Finder

1. Find the package in Finder.
2. Control-click (click while pressing the `ctrl` key) on the package.
3. Select **Open** from the context menu.

From the command line

1. Run `sudo xattr -rd com.apple.quarantine ./Downloads/powershell-7.5.1-osx-x64.pkg`. If you are using PowerShell 7 or higher, you can use the `Unblock-File` cmdlet. Include the full path to the `.pkg` file.
2. Install the package as you normally would.

! Note

This is a known issue related to package notarization that will be addressed in the future.

Install as a .NET Global tool

If you already have the [.NET Core SDK](#) installed, it's easy to install PowerShell as a [.NET Global tool](#).

```
dotnet tool install --global PowerShell
```

The dotnet tool installer adds `~/.dotnet/tools` to your `PATH` environment variable. However, the currently running shell doesn't have the updated `PATH`. You should be able to start PowerShell from a new shell by typing `pwsh`.

Binary Archives

PowerShell binary `tar.gz` archives are provided for the macOS platform to enable advanced deployment scenarios. When you install using this method you must also manually install any dependencies.

! Note

You can use this method to install any version of PowerShell including the latest:

- Stable release: <https://aka.ms/powershell-release?tag=stable> ↗
- LTS release: <https://aka.ms/powershell-release?tag=lts> ↗
- Preview release: <https://aka.ms/powershell-release?tag=preview> ↗

Installing binary archives on macOS

Download the install package from the [releases](#) page onto your computer. The links to the current versions are:

- PowerShell 7.4 (LTS)
 - x64 processors - [powershell-7.4.7-osx-x64.tar.gz](#)
 - Arm64 processors - [powershell-7.4.7-osx-arm64.tar.gz](#)
- PowerShell 7.5-preview
 - x64 processors - [powershell-7.5.1-osx-x64.tar.gz](#)
 - Arm64 processors - [powershell-7.5.1-osx-arm64.tar.gz](#)

Use the following commands to install PowerShell from the binary archive. Change the download URL to match the version you want to install.

```
sh

# Download the powershell '.tar.gz' archive
curl -L -o /tmp/powershell.tar.gz
https://github.com/PowerShell/PowerShell/releases/download/v7.5.1/powershell-
7.5.1-osx-x64.tar.gz

# Create the target folder where powershell is placed
sudo mkdir -p /usr/local/microsoft/powershell/7

# Expand powershell to the target folder
sudo tar zxf /tmp/powershell.tar.gz -C /usr/local/microsoft/powershell/7

# Set execute permissions
sudo chmod +x /usr/local/microsoft/powershell/7/pwsh

# Create the symbolic link that points to pwsh
sudo ln -s /usr/local/microsoft/powershell/7/pwsh /usr/local/bin/pwsh
```

Uninstalling PowerShell

If you installed PowerShell with Homebrew, use the following command to uninstall:

```
sh

brew uninstall --cask powershell
```

If you installed PowerShell via direct download, PowerShell must be removed manually:

```
sh

sudo rm -rf /usr/local/bin/pwsh /usr/local/microsoft/powershell
```

To remove the additional PowerShell paths, refer to the [paths](#) section in this document and remove the paths using `sudo rm`.

 **Note**

This isn't necessary if you installed with Homebrew.

Paths

- `$PSHOME` is `/usr/local/microsoft/powershell/7`
 - The macOS install package creates a symbolic link, `/usr/local/bin/pwsh` that points to `pwsh` in the `$PSHOME` location.
- User profiles are read from `~/.config/powershell/profile.ps1`
- Default profiles are read from `$PSHOME/profile.ps1`
- User modules are read from `~/.local/share/powershell/Modules`
- Shared modules are read from `/usr/local/share/powershell/Modules`
- Default modules are read from `$PSHOME/Modules`
- PSReadLine history are recorded to
`~/.local/share/powershell/PSReadLine/ConsoleHost_history.txt`

PowerShell respects the [XDG Base Directory Specification](#) on macOS.

Supported versions

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of macOS reaches end-of-support.

- macOS 15 (Sequoia) x64 and Arm64
- macOS 14 (Sonoma) x64 and Arm64
- macOS 13 (Ventura) x64 and Arm64

Apple determines the support lifecycle of macOS. For more information, see the following:

- [macOS release notes](#)
- [Apple Security Updates](#)

Installation support

Microsoft supports the installation methods in this document. There may be other methods of installation available from other sources. While those tools and methods may work, Microsoft

can't support those methods.

Additional Resources

- [Homebrew Web ↗](#)
- [Homebrew GitHub Repository ↗](#)
- [Homebrew-Cask ↗](#)

PowerShell on Arm processors

Article • 12/12/2024

Support for the Arm processor is based on the support policy of the version of .NET that PowerShell uses. While .NET supports many more operating systems and versions, PowerShell support is limited to the versions that have been tested.

It may be possible to use Arm-based versions of PowerShell on other Linux distributions and versions, but we don't officially support it.

PowerShell 7.4

Arm versions of PowerShell 7.4 can be installed on the following platforms:

[] Expand table

OS	Architectures	Lifecycle
Windows 11 Client Version 22000+	Arm64	Windows ↗
Windows 10 Client Version 1607+	Arm64	Windows ↗
macOS	Arm64	macOS ↗
Raspberry Pi OS (Debian 12)	Arm32	Raspberry Pi OS ↗ and Debian ↗
Ubuntu 22.04, 20.04	Arm32	Ubuntu ↗

Support is based on the [.NET 8.0 Supported OS Lifecycle Policy](#) ↗.

Installing PowerShell on Arm-based systems

For installation instructions, see the following articles:

Windows

- [Windows 10 on Arm](#)
- [Windows 10 IoT Enterprise](#)
- [Windows 10 IoT Core](#)

Linux - install from the binary archives

- [Alternate ways to install PowerShell on Linux](#)

macOS

- [Installing PowerShell on macOS](#)

Raspberry Pi

- [Raspberry Pi OS](#)

Use PowerShell in Docker

Article • 03/26/2025

The .NET team publishes Docker images with PowerShell preinstalled. This article shows you how to get started using PowerShell in the Docker container.

Find available images

These images require Docker 17.05 or newer. Also, you must be able to run Docker without `sudo` or local administrative rights. For install instructions, see Docker's official [documentation](#).

The .NET team publishes several Docker images designed for different development scenarios. Only the image for the .NET SDK contains PowerShell. For more information, see [Official .NET Docker images](#).

Use PowerShell in a container

The following command downloads the image containing the latest available stable versions of the .NET SDK and PowerShell.

```
Console
```

```
docker pull mcr.microsoft.com/dotnet/sdk:9.0
```

Use the following command to start an interactive PowerShell session in the container.

```
Console
```

```
docker run -it mcr.microsoft.com/dotnet/sdk:9.0 pwsh
```

To download and run the latest Long Term Support (LTS) version of PowerShell, change the image name to `mcr.microsoft.com/dotnet/sdk:8.0`. When you use these image tags, Docker downloads the appropriate image for your host operating system. If you want an image for a specific operating system, you can specify the operating system in the image tag. See the [Microsoft Artifact Registry](#) for a list of available tags.

- For more information about tags, the [Supported tag policy](#)
- For more information about supported operating systems, see the [Supported platforms policy](#)

Support lifecycle

The [.NET support policy](#) defines how these images are supported. These images are provided for development and testing purposes only. If you need a production-ready image, you should build your own images. For more information about these Docker images, visit the [dotnet-docker](#) repository on GitHub.

The images previously published by the PowerShell team will be marked as deprecated in the Microsoft Container Registry (MCR).

Telemetry

By default, PowerShell collects limited telemetry without personal data to help aid development of future versions of PowerShell. To opt-out of sending telemetry, create an environment variable called `POWERSHELL_TELEMETRY_OPTOUT` set to a value of `1` before starting PowerShell from the installed location. The telemetry we collect falls under the [Microsoft Privacy Statement](#).

Microsoft Update for PowerShell FAQ

FAQ

Beginning with PowerShell 7.2, when you install using the MSI package you have the option of enabling Microsoft Update support for PowerShell.

General Information

What is the Microsoft Update feature in PowerShell?

The Microsoft Update feature of PowerShell allows you to get the latest PowerShell 7 updates in your traditional Microsoft Update (MU) management flow, whether that's with Windows Update for Business, WSUS, Microsoft Endpoint Configuration Manager, or the interactive MU dialog in Settings. Microsoft Update and the related services enable you to deploy updates:

- On your schedule
- After testing for your environment
- At scale across your enterprise

How soon after release are updates advertised by Microsoft Update?

When a new version of PowerShell is released, it can take up to two weeks for that version to become available through Microsoft Update. Updates are delivered as optional software updates, even if the update contains a security fix.

If you need to deploy the update before it becomes available in Microsoft Update, download the update from the [Releases](#) page on GitHub.

Why is the latest LTS version not marked as LTS?

We mark the earliest minor version LTS until it goes out of support. For example, both PowerShell 7.2 and 7.4 are LTS releases and have a year of overlapping support. PowerShell 7.2 was marked as the latest LTS in MU until it reached end of support in November 2024.

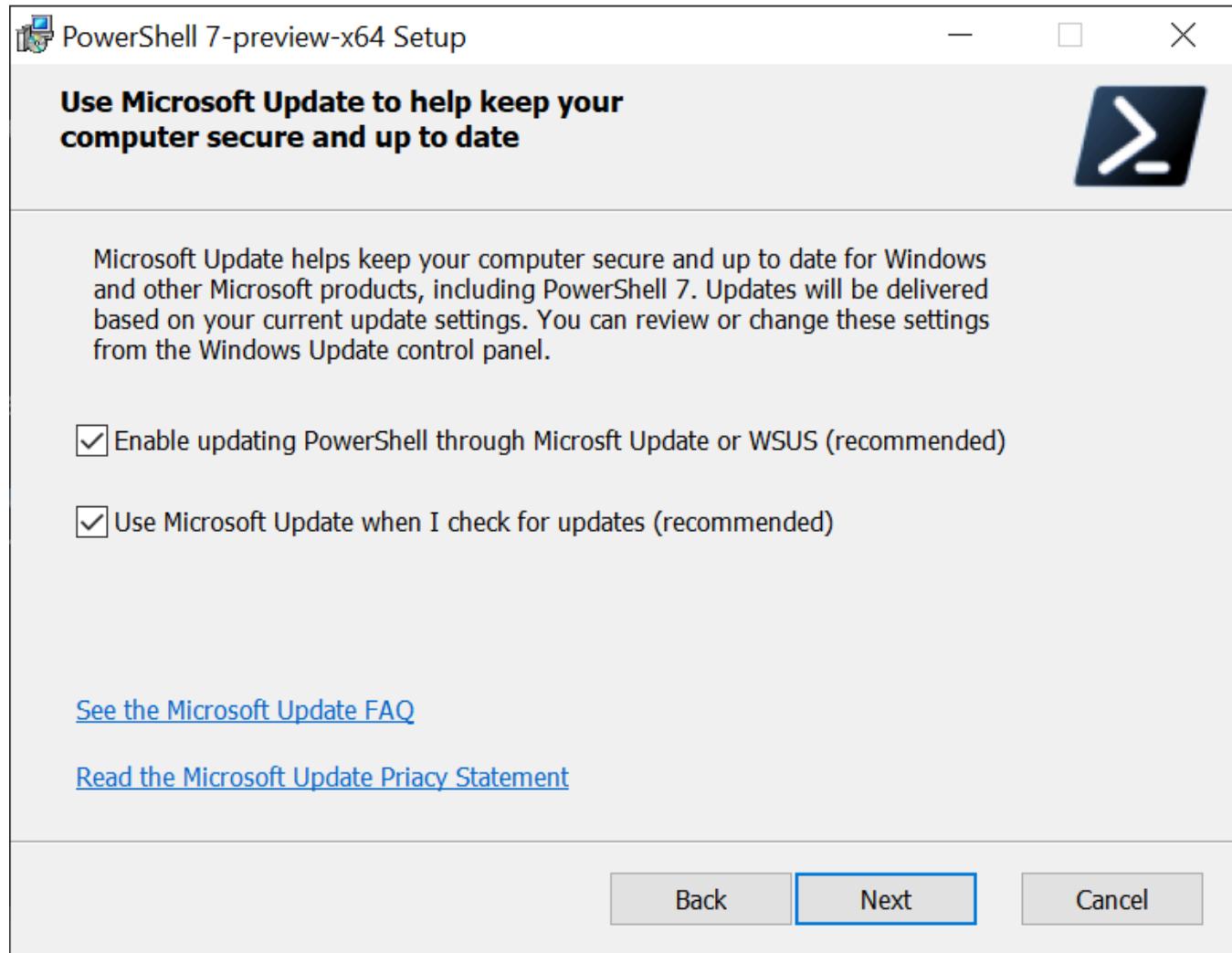
Configuration

What version of Windows is required to support the Microsoft Update feature?

You must have Windows Version 1809 or newer installed on an x64-based system. Version 1809 is the Windows 10 October 2018 Update or Windows Server 2019. Versions prior to 1809 do not support Microsoft Update for PowerShell.

Do I need to check both boxes in the setup dialog?

While the two options on the dialog are independent, in most cases, it's best to check both boxes.



What does each checkbox do?

The first checkbox enables updates for PowerShell. These updates can be delivered by Microsoft Update, a WSUS server, or SCCM. If this checkbox is unchecked, you cannot receive updates through any of these channels.

The second checkbox enables Microsoft Update on your system. This allows you to receive updates for any supported Microsoft software, not just Windows. If the box is unchecked, you will not receive the update from Microsoft Update, but you can receive updates from WSUS or SCCM.

What if I want to opt-out later?

If you want to opt-out of updates later, you can run the MSI install package and uncheck the first checkbox. Unchecking the second checkbox has no effect.

Can I enable these update options from the command line or in a script?

Yes. The MSI package includes two new MSI options for enabling the update features:

- `USE_MU` - This property has two possible values:
 - `1` (default) - Opts into updating through Microsoft Update, WSUS, or SCCM
 - `0` - Do not opt into updating through Microsoft Update, WSUS, or SCCM
- `ENABLE_MU`
 - `1` (default) - Opts into using Microsoft Update for Automatic Updates
 - `0` - Do not opt into using Microsoft Update

Note

Setting `ENABLE_MU=0` does not disable Microsoft Update.

Troubleshooting

Why haven't I received an update for the new release?

There can be several reasons for not receiving the update:

- We may not have published the update yet. Our goal is to make the update available to Microsoft Update within two weeks of release, but there is no guarantee for that availability.
- There are group policy settings that control Microsoft Update. Your system administrator may have policies set that prevent you from using Microsoft Update. The checkbox in the installer cannot override the Group Policy.
- Make sure you have checked both checkboxes. When doing a repair installation, the installer doesn't show the check box options. To enable MU updates run the following command:

PowerShell

```
msiexec.exe /fmu .\PowerShell-7.4.10-win-x64.msi USE_MU=1 ENABLE_MU=1
```

For more information about running `msiexec.exe` from the command line, see [msiexec](#).

I am on PowerShell 7.x, why have I not been upgraded to 7.y?

The Microsoft Update feature for PowerShell only updates versions in the same release channel. PowerShell 7.4 is the latest long term supported (LTS) version. PowerShell 7.5 is the latest stable (non-LTS) version. Microsoft Update provides updates for the next patch level versions of either version. For example:

- If you are running 7.4, you will receive updates for 7.4.
- If you are running 7.5, you will receive updates for 7.5.

Microsoft Update will never upgrade an LTS release to a stable non-LTS release. However, a stable non-LTS release will be upgraded to the higher LTS release when support for the stable release ends.

Preview releases are never upgraded to the GA release version. However, they will be upgraded to the next available preview release. For example: Consider the scenario where you have 7.4 (LTS) installed and the 7.5-rc.1 (preview) release installed. When 7.5.0 (Stable) released, your 7.4 (LTS) installation is not upgraded to 7.5.0. Also, 7.5.0 can't upgrade 7.5-rc.1. However, 7.6-preview.2 can upgrade 7.5-rc.1.

For more information, see [PowerShell Support Lifecycle](#).

Introduction

Article • 06/25/2024

 Expand table



This content originally appeared in the book *PowerShell 101* by Mike F. Robbins. We thank Mike for granting us permission to reuse the content here. We edited the content for publication on this platform. You can still get the original book from Leanpub at [PowerShell 101](#).

About this book

Before PowerShell, I began my career as an IT Pro, pointing and clicking in the GUI. I wrote this book to save IT Pros from themselves by reducing the learning curve and helping them avoid being reluctant to learn PowerShell.

Instead of a book that covers topics with fictitious scenarios, this book is a condensed version targeting the specific topics that an IT Pro needs to know to succeed with PowerShell in a real-world production environment. It's a collection of what I wish someone had told me when I started learning PowerShell. I include tips, tricks, and best practices that I learned while using PowerShell since 2007.

Each chapter includes a curated collection of links to specific help articles that expand on the information covered. These resources expand on the concepts discussed and broaden your understanding of PowerShell.

Who is this book for?

This book is for anyone wanting to learn PowerShell. Whether you're a beginner or an experienced user, this book helps you improve your PowerShell skills.

This book focuses on Windows PowerShell version 5.1 running on Windows 11 and Windows Server 2022 in a Microsoft Active Directory domain environment. However, the basic concepts apply to all versions of PowerShell running on any supported platform.

Lab environment

The examples in this book were created and tested on Windows 11 and Windows Server 2022 operating systems, using Windows PowerShell version 5.1. If you're running a

different operating system or version of PowerShell, your results might vary from the ones presented in this book.

About the author

Mike F. Robbins, a former Microsoft MVP, is the lead technical writer for [Azure PowerShell](#) at Microsoft. With extensive experience in PowerShell, he is a scripting, automation, and efficiency expert. As a lifelong learner, Mike continuously strives to improve his skills and empower others by sharing his knowledge and experience. He is also a published author of several books, including:

- Author of [PowerShell 101: The No-Nonsense Guide to Windows PowerShell](#)
- Creator of [The PowerShell Conference Book](#)
- Coauthor of [Windows PowerShell TFM 4th Edition](#)
- Contributing author in the [PowerShell Deep Dives](#) book

When Mike's not writing documentation for Microsoft, he shares his thoughts and insights on his blog at [mikefrobbins.com](#) and interacts with his followers on Twitter @mikefrobbins.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Chapter 1 - Getting started with PowerShell

Article • 08/02/2024

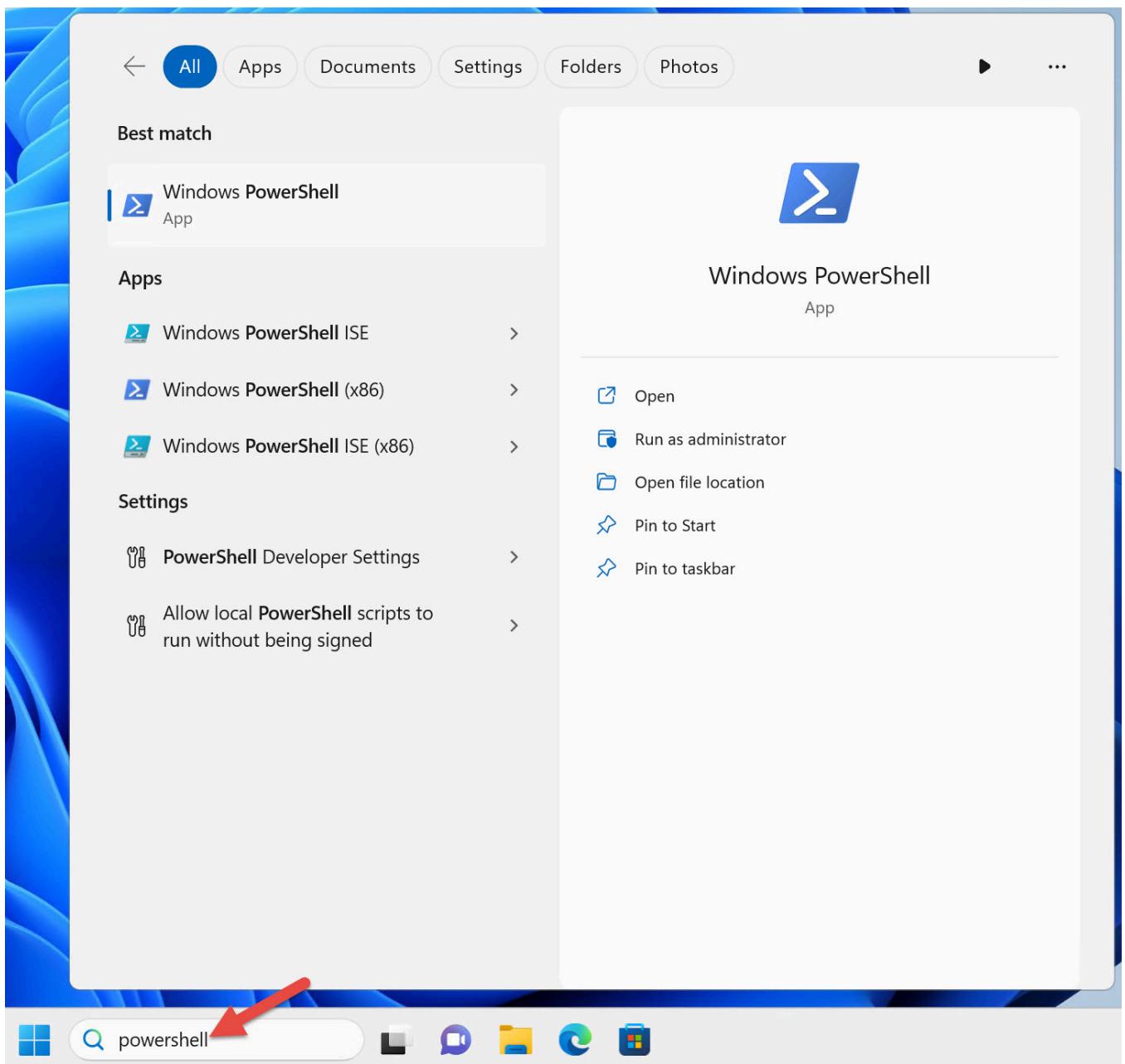
This chapter focuses on finding and launching PowerShell and solving the initial pain points that new users experience with PowerShell. Follow along and walk through the examples in this chapter on your lab environment computer.

What is PowerShell?

Windows PowerShell is an easy-to-use command-line shell and scripting environment for automating administrative tasks of Windows-based systems. Windows PowerShell is preinstalled on all modern versions of the Windows operating system.

Where to find PowerShell

The easiest way to find PowerShell on Windows 11 is to type `PowerShell` into the search bar, as shown in Figure 1-1. Notice that there are four different shortcuts for Windows PowerShell.



Windows PowerShell shortcuts on a 64-bit version of Windows:

- Windows PowerShell
- Windows PowerShell ISE
- Windows PowerShell (x86)
- Windows PowerShell ISE (x86)

On a 64-bit version of Windows, you have a 64-bit version of the Windows PowerShell console and the Windows PowerShell Integrated Scripting Environment (ISE) and a 32-bit version of each one, as indicated by the (x86) suffix on the shortcuts.

! Note

Windows 11 only ships as a 64-bit operating system. There is no 32-bit version of Windows 11. However, Windows 11 includes 32-bit versions of Windows PowerShell and the Windows PowerShell ISE.

You only have two shortcuts if you're running an older 32-bit version of Windows. Those shortcuts don't have the (x86) suffix but are 32-bit versions.

I recommend using the 64-bit version of Windows PowerShell if you're running a 64-bit operating system unless you have a specific reason for using the 32-bit version.

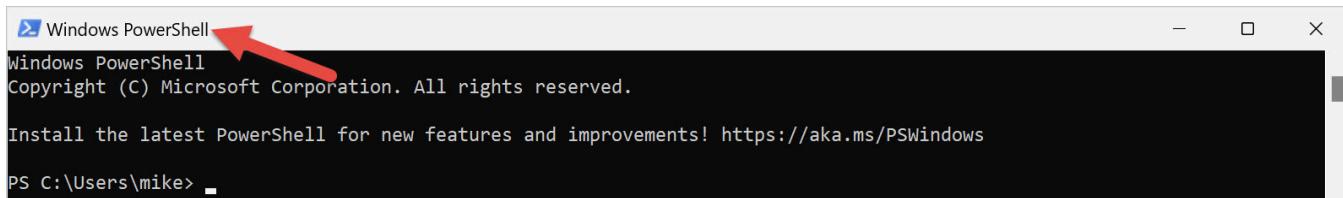
Depending on what version of Windows 11 you're running, Windows PowerShell might open in [Windows Terminal](#).

Microsoft no longer updates the PowerShell ISE. The ISE only works with Windows PowerShell 5.1. [Visual Studio Code](#) (VS Code) with the [PowerShell extension](#) works with both versions of PowerShell. VS Code and the PowerShell extension don't ship in Windows. Install VS Code and the extension on the computer where you create PowerShell scripts. You don't need to install them on all the computers where you run PowerShell.

How to launch PowerShell

I use three different Active Directory user accounts in the production environments I support. I mirrored those accounts in the lab environment used in this book. I sign into my Windows 11 computer as a domain user without domain or local administrator rights.

Launch the PowerShell console by clicking the **Windows PowerShell** shortcut, as shown in Figure 1-1. Notice that the title bar of the console says **Windows PowerShell**, as shown in Figure 1-2.



Some commands run fine when you run PowerShell as an ordinary user. However, PowerShell doesn't participate in [User Access Control \(UAC\)](#). That means it's unable to prompt for elevation for tasks that require the approval of an administrator.

⚠ Note

UAC is a Windows security feature that helps prevent malicious code from running with elevated privileges.

When signed on as an ordinary user, PowerShell returns an error when you run a command that requires elevation. For example, stopping a Windows service:

```
PowerShell

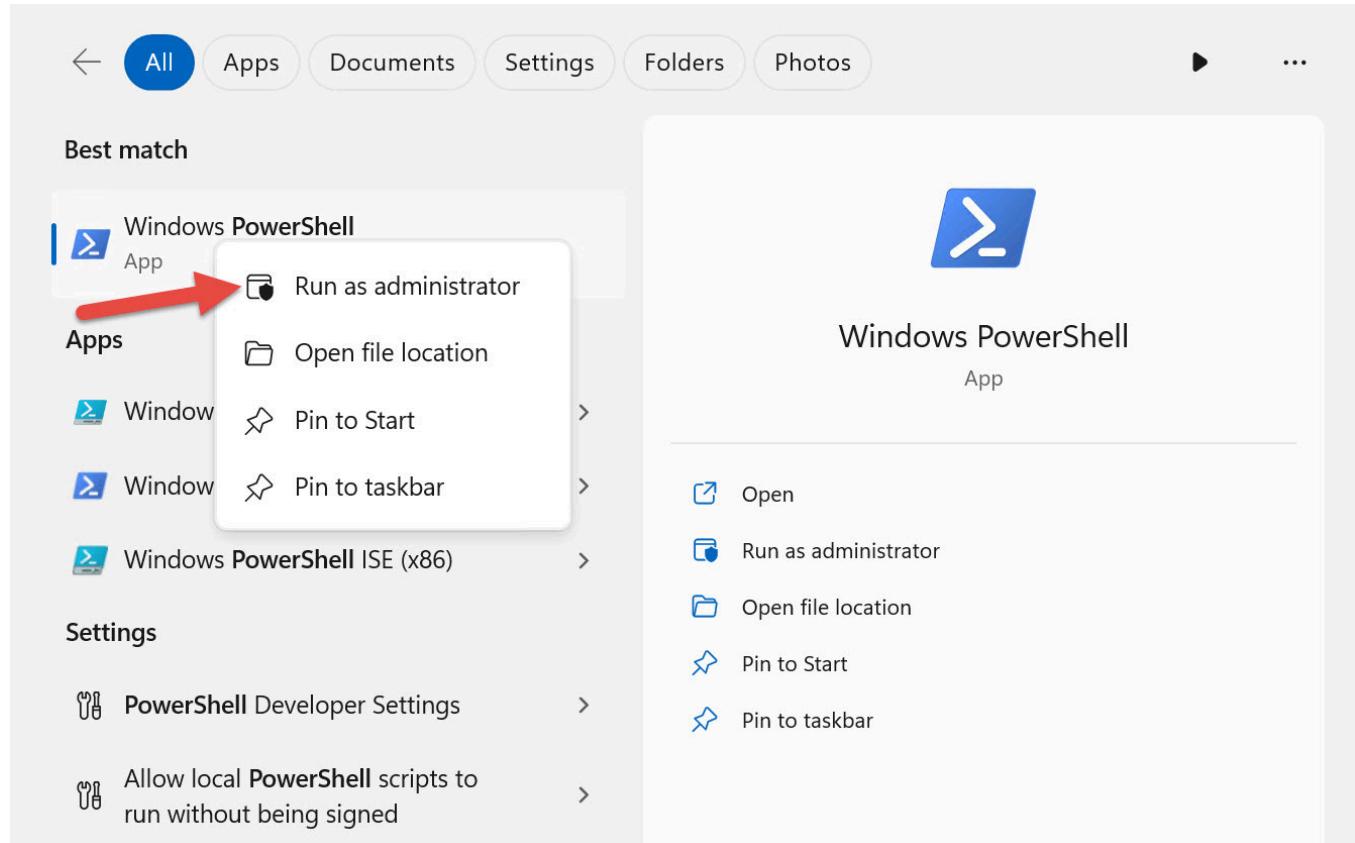
Stop-Service -Name W32Time

Output

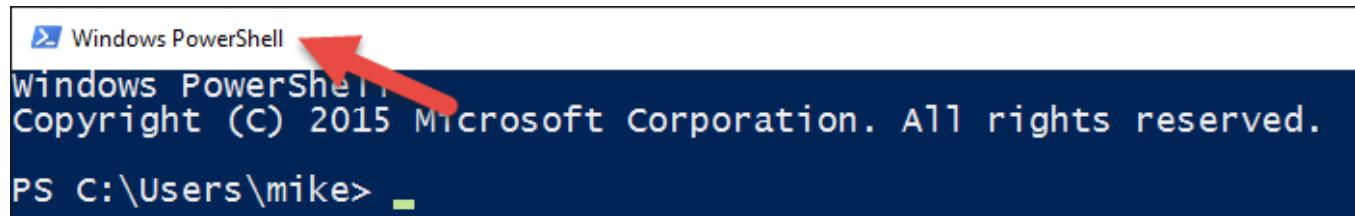
Stop-Service : Service 'Windows Time (W32Time)' cannot be stopped due to
the following error: Cannot open W32Time service on computer '.'.
At line:1 char:1
+ Stop-Service -Name W32Time
+ ~~~~~
+ CategoryInfo          : CloseError: (System.ServiceProcess.ServiceCon
troller:ServiceController) [Stop-Service], ServiceCommandException
+ FullyQualifiedErrorId : CouldNotStopService,Microsoft.PowerShell.Com
ands.StopServiceCommand
```

The solution is to run PowerShell elevated as a user who is a local administrator. That's how I configured my second domain user account. Following the principle of least privilege, this account shouldn't be a domain administrator or have any elevated privileges in the domain.

To start PowerShell with elevated rights, right-click the **Windows PowerShell** shortcut and select **Run as administrator**, as shown in Figure 1-3.

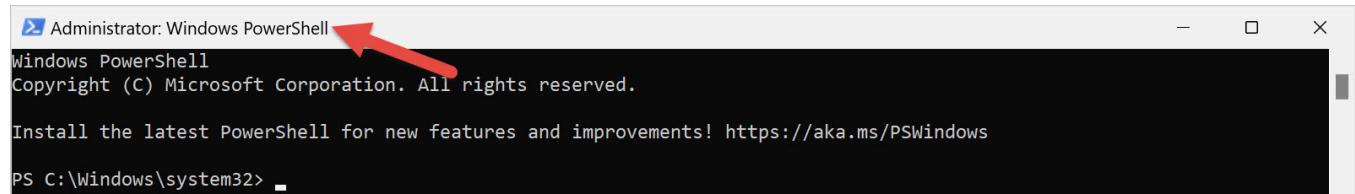


Windows prompts you for credentials because you logged into Windows as an ordinary user. Enter the credentials of your domain user who is a local administrator, as shown in Figure 1-4.



A screenshot of a Windows PowerShell window. The title bar says "Windows PowerShell". The content area shows the PowerShell prompt "PS C:\Users\mike>". A red arrow points to the title bar.

Notice that the title bar of the elevated console windows says **Administrator: Windows PowerShell**, as shown in Figure 1-5.



A screenshot of an elevated Windows PowerShell window. The title bar says "Administrator: Windows PowerShell". The content area shows the PowerShell prompt "PS C:\Windows\system32>". A red arrow points to the title bar.

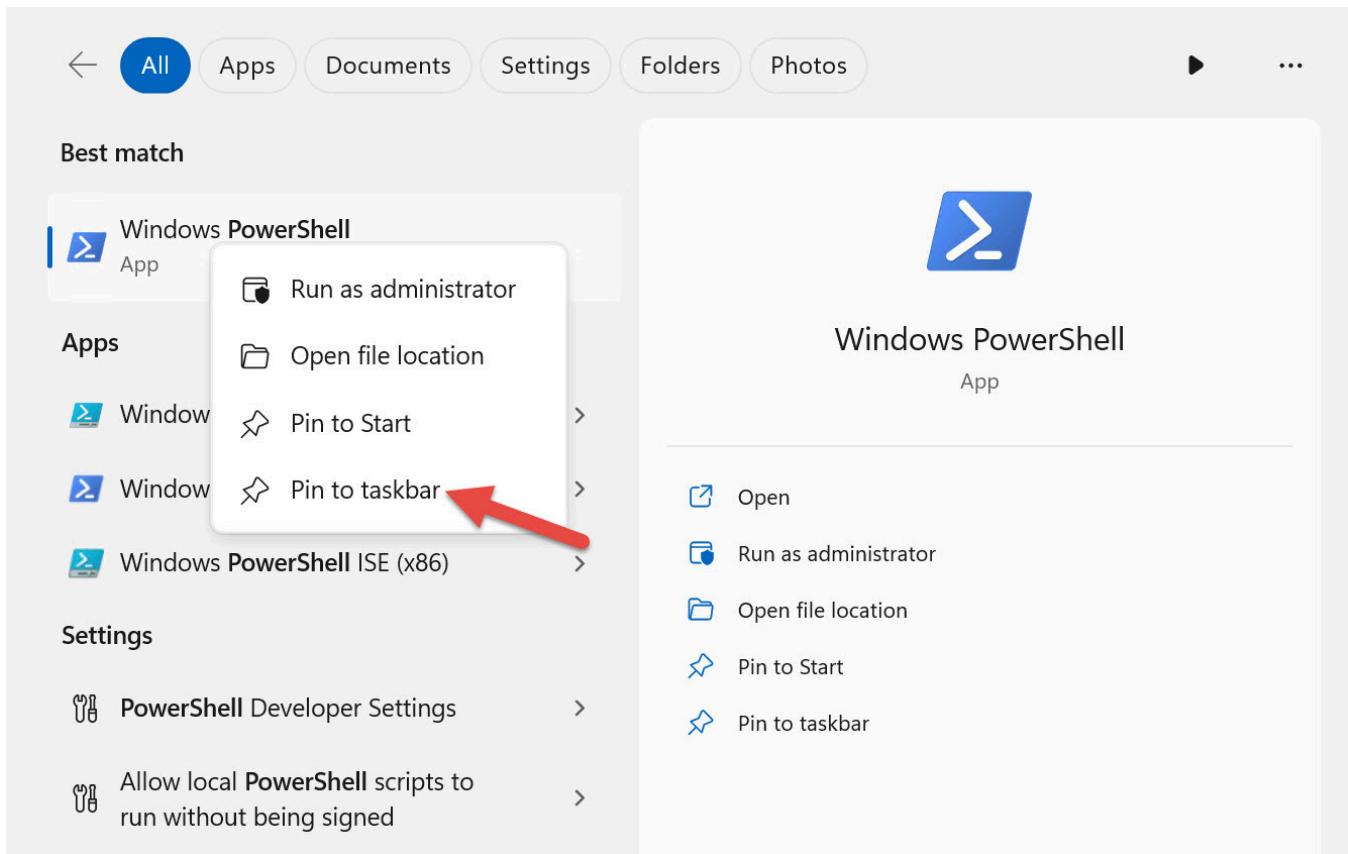
Now that you're running PowerShell elevated as an administrator, UAC is no longer a problem when you run a command that requires elevation.

 **Important**

You should only run PowerShell elevated as an administrator when absolutely necessary.

When you target remote computers, there's no need to run PowerShell elevated. Running PowerShell elevated only affects commands that run against your local computer.

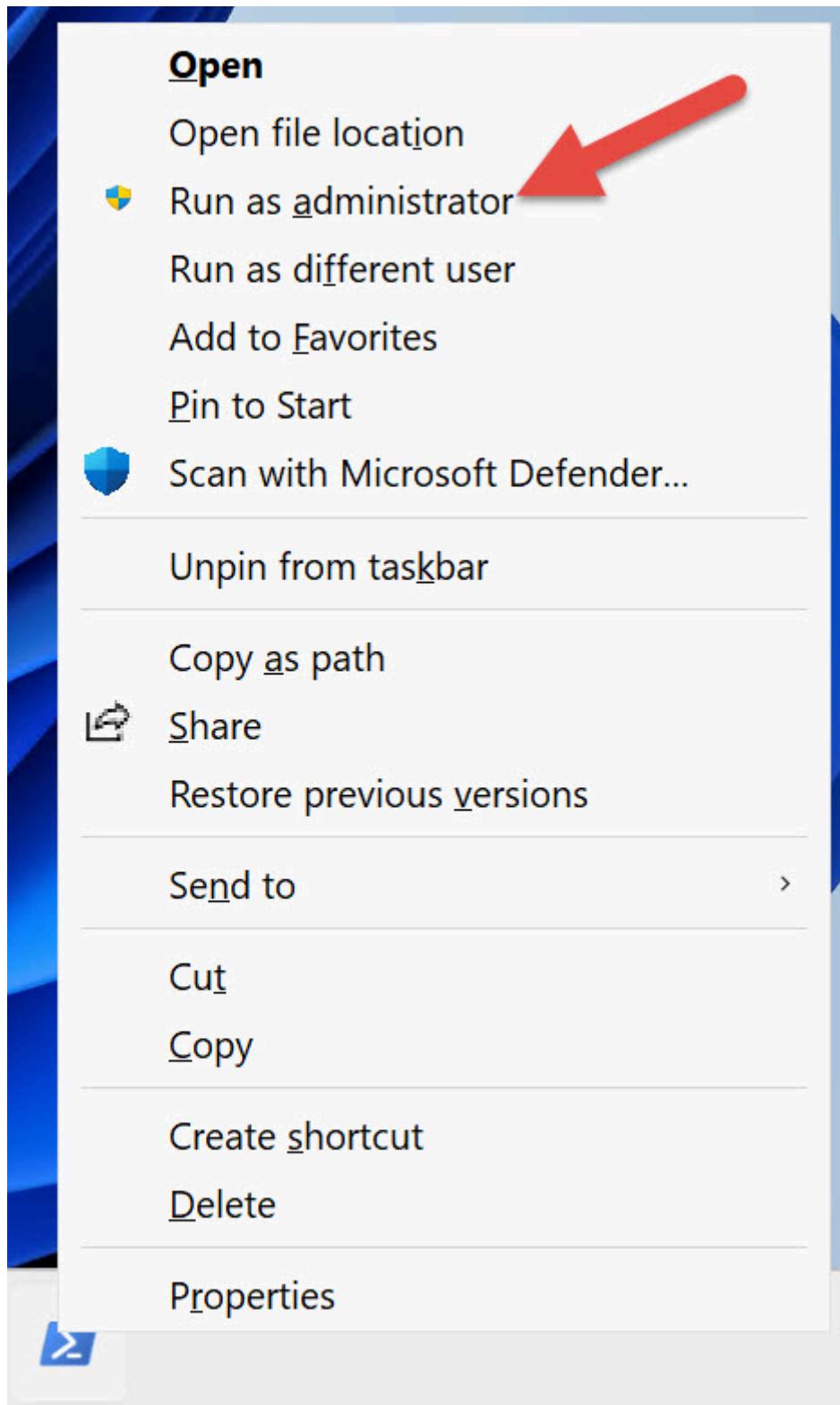
You can simplify finding and launching PowerShell. Pin the PowerShell or Windows Terminal shortcut to your taskbar. Search for PowerShell again, except this time right-click on it and select **Pin to taskbar** as shown in Figure 1-6.



ⓘ Important

The original version of this book, published in 2017, recommended pinning a shortcut to the taskbar to launch an elevated instance automatically every time you start PowerShell. However, due to potential security concerns, I no longer recommend it. Any applications you launch from an elevated instance of PowerShell also bypass UAC and run elevated. For example, if you launch a web browser from an elevated instance of PowerShell, any website you visit containing malicious code also runs elevated.

When you need to run PowerShell with elevated permissions, right-click the PowerShell shortcut pinned to your taskbar while pressing `Shift`. Select **Run as administrator**, as shown in Figure 1-7.



Determine your version of PowerShell

There are automatic variables in PowerShell that store state information. One of these variables is `$PSVersionTable`, which contains version information about your PowerShell session.

```
PowerShell
$PSVersionTable
```

Output

Name	Value
PSVersion	5.1.22621.2428
PSEdition	Desktop
PSCompatibleVersions	{1.0, 2.0, 3.0, 4.0...}
BuildVersion	10.0.22621.2428
CLRVersion	4.0.30319.42000
WSManStackVersion	3.0
PSRemotingProtocolVersion	2.3
SerializationVersion	1.1.0.1

If you're running a version of Windows PowerShell older than 5.1, you should update your version of Windows. Windows PowerShell 5.1 is preinstalled on the currently supported versions of Windows.

PowerShell version 7 isn't a replacement for Windows PowerShell 5.1; it installs side-by-side with Windows PowerShell. Windows PowerShell version 5.1 and PowerShell version 7 are two different products. For more information about the differences between Windows PowerShell version 5.1 and PowerShell version 7, see [Migrating from Windows PowerShell 5.1 to PowerShell 7](#).

Tip

PowerShell version 6, formerly known as PowerShell Core, is no longer supported.

Execution policy

PowerShell execution policy controls the conditions under which you can run PowerShell scripts. The execution policy in PowerShell is a safety feature designed to help prevent the unintentional execution of malicious scripts. However, it's not a security boundary because it can't stop determined users from deliberately running scripts. A determined user can bypass the execution policy in PowerShell.

You can set an execution policy for the local computer, current user, or a PowerShell session. You can also set execution policies for users and computers with Group Policy.

The following table shows the default execution policy for current Windows operating systems.

Expand table

Windows Operating System Version	Default Execution Policy
Windows Server 2022	Remote Signed
Windows Server 2019	Remote Signed
Windows Server 2016	Remote Signed
Windows 11	Restricted
Windows 10	Restricted

Regardless of the execution policy setting, you can run any PowerShell command interactively. The execution policy only affects commands running in a script. Use the `Get-ExecutionPolicy` cmdlet to determine the current execution policy setting.

Check the execution policy setting on your computer.

```
PowerShell
Get-ExecutionPolicy

Output
Restricted
```

List the execution policy settings for all scopes.

```
PowerShell
Get-ExecutionPolicy -List

Output
Scope ExecutionPolicy
-----
MachinePolicy      Undefined
UserPolicy         Undefined
Process           Undefined
CurrentUser        Undefined
LocalMachine       Undefined
```

All Windows client operating systems have the default execution policy setting of `Restricted`. You can't run PowerShell scripts using the `Restricted` execution policy setting. To test the execution policy, save the following code as a `.ps1` file named `Get-TimeService.ps1`.

Tip

A PowerShell script is a plaintext file that contains the commands you want to run.

PowerShell script files use the `.ps1` file extension. To create a PowerShell script, use a code editor like Visual Studio Code (VS Code) or any text editor such as Notepad.

When you run the following command interactively, it completes without error.

```
PowerShell
```

```
Get-Service -Name W32Time
```

However, PowerShell returns an error when you run the same command from a script.

```
PowerShell
```

```
.\Get-TimeService.ps1
```

```
Output
```

```
.\Get-TimeService.ps1 : File C:\tmp\Get-TimeService.ps1 cannot be loaded
because running scripts is disabled on this system. For more information,
see about_Execution_Policies at
https://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ .\Get-TimeService.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: () [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
```

When you run a command in PowerShell that generates an error, read the error message before retrying the command. Notice the error message tells you why the command failed:

... running scripts is disabled on this system.

To enable the execution of scripts, change the execution policy with the `Set-ExecutionPolicy` cmdlet. `LocalMachine` is the default scope when you don't specify the `Scope` parameter. You must run PowerShell elevated as an administrator to change the execution policy for the local machine. Unless you're signing your scripts, I recommend using the `RemoteSigned` execution policy. `RemoteSigned` prevents you from running downloaded scripts that aren't signed by a trusted publisher.

Before you change the execution policy, read the [about_Execution_Policies](#) help article to understand the security implications.

Change the execution policy setting on your computer to `RemoteSigned`.

PowerShell

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

If you have successfully changed the execution policy, PowerShell displays the following warning:

Output

Execution Policy Change

The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose you to the security risks described in the [about_Execution_Policies](#) help topic at <https://go.microsoft.com/fwlink/?LinkID=135170>. Do you want to change the execution policy?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "N"):`y`

If you're not running PowerShell elevated as an administrator, PowerShell returns the following error message:

Output

```
Set-ExecutionPolicy : Access to the registry key 'HKEY_LOCAL_MACHINE\SOFTWAR
E\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell' is denied. To
change the execution policy for the default (LocalMachine) scope, start
Windows PowerShell with the "Run as administrator" option. To change the
execution policy for the current user, run "Set-ExecutionPolicy -Scope
CurrentUser".
At line:1 char:1
+ Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
+ ~~~~~
+ CategoryInfo          : PermissionDenied: (:) [Set-ExecutionPolicy],
UnauthorizedAccessException
+ FullyQualifiedErrorId : System.UnauthorizedAccessException,Microsoft.
PowerShell.Commands.SetExecutionPolicyCommand
```

It's also possible to change the execution policy for the current user without requiring you to run PowerShell elevated as an administrator. This step is unnecessary if you successfully set the execution policy for the local machine to `RemoteSigned`.

PowerShell

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

With the execution policy set to `RemoteSigned`, the `Get-TimeService.ps1` script runs successfully.

PowerShell

```
.\Get-TimeService.ps1
```

Output

Status	Name	DisplayName
-----	-----	-----
Running	W32Time	Windows Time

Summary

In this chapter, you learned where to find and how to launch PowerShell. You also learned how to determine the version of PowerShell and the purpose of execution policies.

Review

1. How do you determine what PowerShell version a computer is running?
2. When should you launch PowerShell elevated as an administrator?
3. What's the default execution policy on Windows client computers, and what does it prevent you from doing?
4. How do you determine the current PowerShell execution policy setting?
5. How do you change the PowerShell execution policy?

References

To learn more about the concepts covered in this chapter, read the following PowerShell help articles.

- [about_Automatic_Variables](#)
- [about_Execution_Policies](#)

Next steps

In the next chapter, you'll learn about the discoverability of commands in PowerShell. You'll also learn how to download PowerShell's help files so you can view the help in your PowerShell session.

Chapter 2 - The Help system

Article • 06/27/2024

In an experiment designed to assess proficiency in PowerShell, two distinct groups of IT professionals — beginners and experts — were first given a written examination without access to a computer. Surprisingly, the test scores indicated comparable skills across both groups. A subsequent test was then administered, mirroring the first but with one key difference: participants had access to an offline computer equipped with PowerShell. The results revealed a significant skills gap between the two groups this time.

What factors contributed to the outcomes observed between the two assessments?

Experts don't always know the answers, but they know how to figure out the answers.

The outcomes observed in the results of the two tests were because experts don't memorize thousands of PowerShell commands. Instead, they excel at using the Help system within PowerShell, enabling them to discover and learn how to use commands when necessary.

Becoming proficient with the Help system is the key to success with PowerShell.

I heard Jeffrey Snover, the creator of PowerShell, share a similar story on multiple occasions.

Discoverability

Compiled commands in PowerShell are known as cmdlets, pronounced as "*command-let*", not "*CMD-let*". The naming convention for cmdlets follows a singular **Verb-Noun** format to make them easily discoverable. For instance, `Get-Process` is the cmdlet to determine what processes are running, and `Get-Service` is the cmdlet to retrieve a list of services. Functions, also known as script cmdlets, and aliases are other types of PowerShell commands that are discussed later in this book. The term "*PowerShell command*" describes any command in PowerShell, regardless of whether it's a cmdlet, function, or alias.

You can also run operating system native commands from PowerShell, such as traditional command-line programs like `ping.exe` and `ipconfig.exe`.

The three core cmdlets in PowerShell

- `Get-Help`
- `Get-Command`
- `Get-Member` (covered in chapter 3)

I'm often asked: "*How do you figure out what the commands are in PowerShell?*". Both `Get-Help` and `Get-Command` are invaluable resources for discovering and understanding commands in PowerShell.

Get-Help

The first thing you need to know about the Help system in PowerShell is how to use the `Get-Help` cmdlet.

`Get-Help` is a multipurpose command that helps you learn how to use commands once you find them. You can also use `Get-Help` to locate commands, but in a different and more indirect way when compared to `Get-Command`.

When using `Get-Help` to locate commands, it initially performs a wildcard search for command names based on your input. If that doesn't find any matches, it conducts a comprehensive full-text search across all PowerShell help articles on your system. If that also fails to find any results, it returns an error.

Here's how to use `Get-Help` to view the help content for the `Get-Help` cmdlet.

```
PowerShell
Get-Help -Name Get-Help
```

Beginning with PowerShell version 3.0, the help content doesn't ship preinstalled with the operating system. When you run `Get-Help` for the first time, a message asks if you want to download the PowerShell help files to your computer.

Answering **Yes** by pressing `y` executes the `Update-Help` cmdlet, downloading the help content.

```
Output
Do you want to run Update-Help?
The Update-Help cmdlet downloads the most current Help files for Windows
PowerShell modules, and installs them on your computer. For more information
about the Update-Help cmdlet, see
```

```
https://go.microsoft.com/fwlink/?LinkId=210614.  
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"):
```

If you don't receive this message, run `Update-Help` from an elevated PowerShell session running as an administrator.

Once the update is complete, the help article is displayed.

Take a moment to run the example on your computer, review the output, and observe how the help system organizes the information.

- NAME
- SYNOPSIS
- SYNTAX
- DESCRIPTION
- RELATED LINKS
- REMARKS

As you review the output, keep in mind that help articles often contain a vast amount of information, and what you see by default isn't the entire help article.

Parameters

When you run a command in PowerShell, you might need to provide additional information or input to the command. Parameters allow you to specify options and arguments that change the behavior of a command. The **SYNTAX** section of each help article outlines the available parameters for the command.

`Get-Help` has several parameters that you can specify to return the entire help article or a subset for a command. To view all the available parameters for `Get-Help`, see the **SYNTAX** section of its help article, as shown in the following example.

Output

```
...  
SYNTAX  
    Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider  
    | General | FAQ | Glossary | HelpFile | ScriptCommand | Function |  
    Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |  
    Class | Configuration}] [-Component <System.String[]>] [-Full]  
    [-Functionality <System.String[]>] [-Path <System.String>] [-Role  
    <System.String[]>] [<CommonParameters>]  
  
    Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider  
    | General | FAQ | Glossary | HelpFile | ScriptCommand | Function |  
    Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |
```

```
Class | Configuration}] [-Component <System.String[]>] -Detailed  
[-Functionality <System.String[]>] [-Path <System.String>] [-Role  
<System.String[]>] [<CommonParameters>]  
  
Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider  
| General | FAQ | Glossary | HelpFile | ScriptCommand | Function |  
Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |  
Class | Configuration}] [-Component <System.String[]>] -Examples  
[-Functionality <System.String[]>] [-Path <System.String>] [-Role  
<System.String[]>] [<CommonParameters>]  
  
Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider  
| General | FAQ | Glossary | HelpFile | ScriptCommand | Function |  
Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |  
Class | Configuration}] [-Component <System.String[]>] [-Functionality  
<System.String[]>] -Online [-Path <System.String>] [-Role  
<System.String[]>] [<CommonParameters>]  
  
Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider  
| General | FAQ | Glossary | HelpFile | ScriptCommand | Function |  
Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |  
Class | Configuration}] [-Component <System.String[]>] [-Functionality  
<System.String[]>] -Parameter <System.String> [-Path <System.String>]  
[-Role <System.String[]>] [<CommonParameters>]  
  
Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider  
| General | FAQ | Glossary | HelpFile | ScriptCommand | Function |  
Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |  
Class | Configuration}] [-Component <System.String[]>] [-Functionality  
<System.String[]>] [-Path <System.String>] [-Role <System.String[]>]  
-ShowWindow [<CommonParameters>]  
...
```

Parameter sets

When you review the **SYNTAX** section for `Get-Help`, notice that the information appears to be repeated six times. Each of those blocks is an individual parameter set, indicating the `Get-Help` cmdlet features six distinct sets of parameters. A closer look reveals each parameter set contains at least one unique parameter, making it different from the others.

Parameter sets are mutually exclusive. Once you specify a unique parameter that only exists in one parameter set, PowerShell limits you to using the parameters contained within that parameter set. For instance, you can't use the **Full** and **Detailed** parameters of `Get-Help` together because they belong to different parameter sets.

Each of the following parameters belongs to a different parameter set for the `Get-Help` cmdlet.

- Full
- Detailed
- Examples
- Online
- Parameter
- ShowWindow

The command syntax

If you're new to PowerShell, comprehending the cryptic information — characterized by square and angle brackets — in the **SYNTAX** section might seem overwhelming. However, learning these syntax elements is essential to becoming proficient with PowerShell. The more frequently you use the PowerShell Help system, the easier it becomes to remember all the nuances.

View the syntax of the `Get-EventLog` cmdlet.

```
PowerShell

Get-Help Get-EventLog
```

The following output shows the relevant portion of the help article.

```
Output

...
SYNTAX
  Get-EventLog [-LogName] <System.String> [[-InstanceId]
    <System.Int64[]>] [-After <System.DateTime>] [-AsBaseObject] [-Before
    <System.DateTime>] [-ComputerName <System.String[]>] [-EntryType {Error
    | Information | FailureAudit | SuccessAudit | Warning}] [-Index
    <System.Int32[]>] [-Message <System.String>] [-Newest <System.Int32>]
    [-Source <System.String[]>] [-UserName <System.String[]>]
    [<CommonParameters>]

  Get-EventLog [-AsString] [-ComputerName <System.String[]>] [-List]
    [<CommonParameters>]
...
```

The syntax information includes pairs of square brackets (`[]`). Depending on their usage, these square brackets serve two different purposes.

- Elements enclosed in square brackets are optional.
- An empty set of square brackets following a datatype, such as `<string[]>`, indicates that the parameter can accept multiple values passed as an array or a

collection object.

Positional parameters

Some cmdlets are designed to accept positional parameters. Positional parameters allow you to provide a value without specifying the name of the parameter. When using a parameter positionally, you must specify its value in the correct position on the command line. You can find the positional information for a parameter in the **PARAMETERS** section of a command's help article. When you explicitly specify parameter names, you can use the parameters in any order.

For the `Get-EventLog` cmdlet, the first parameter in the first parameter set is **LogName**. **LogName** is enclosed in square brackets, indicating it's a positional parameter.

Syntax

```
Get-EventLog [-LogName] <System.String>
```

Since **LogName** is a positional parameter, you can specify it by either name or position. According to the angle brackets following the parameter name, the value for **LogName** must be a single string. The absence of square brackets enclosing both the parameter name and datatype indicates that **LogName** is a required parameter within this particular parameter set.

The second parameter in that parameter set is **InstanceId**. Both the parameter name and datatype are entirely enclosed in square brackets, signifying that **InstanceId** is an optional parameter.

Syntax

```
[[-InstanceId] <System.Int64[]>]
```

Furthermore, **InstanceId** has its own pair of square brackets, indicating that it's a positional parameter similar to the **LogName** parameter. Following the datatype, an empty set of square brackets implies that **InstanceId** can accept multiple values.

Switch parameters

A parameter that doesn't require a value is called a switch parameter. You can easily identify switch parameters because there's no datatype following the parameter name. When you specify a switch parameter, its value is `true`. When you don't specify a switch parameter, its value is `false`.

The second parameter set includes a **List** parameter, which is a switch parameter. When you specify the **List** parameter, it returns a list of event logs on the local computer.

Syntax

```
[-List]
```

A simplified approach to syntax

There's a more user-friendly method to obtain the same information as the cryptic command syntax for some commands, except in plain English. PowerShell returns the complete help article when using `Get-Help` with the **Full** parameter, making it easier to understand a command's usage.

PowerShell

```
Get-Help -Name Get-Help -Full
```

Take a moment to run the example on your computer, review the output, and observe how the help system organizes the information.

- NAME
- SYNOPSIS
- SYNTAX
- DESCRIPTION
- PARAMETERS
- INPUTS
- OUTPUTS
- NOTES
- EXAMPLES
- RELATED LINKS

By specifying the **Full** parameter with the `Get-Help` cmdlet, the output includes several extra sections. Among these sections, **PARAMETERS** often provides a detailed explanation for each parameter. However, the extent of this information varies depending on the specific command you're investigating.

Output

```
...
-Detailed <System.Management.Automation.SwitchParameter>
    Adds parameter descriptions and examples to the basic help display.
    This parameter is effective only when the help files are installed
```

on the computer. It has no effect on displays of conceptual (`About_`) help.

Required?	true
Position?	named
Default value	False
Accept pipeline input?	False
Accept wildcard characters?	false

-Examples <System.Management.Automation.SwitchParameter>

Displays only the name, synopsis, and examples. This parameter is effective only when the help files are installed on the computer. It has no effect on displays of conceptual (`About_`) help.

Required?	true
Position?	named
Default value	False
Accept pipeline input?	False
Accept wildcard characters?	false

-Full <System.Management.Automation.SwitchParameter>

Displays the entire help article for a cmdlet. Full includes parameter descriptions and attributes, examples, input and output object types, and additional notes.

This parameter is effective only when the help files are installed on the computer. It has no effect on displays of conceptual (`About_`) help.

Required?	false
Position?	named
Default value	False
Accept pipeline input?	False
Accept wildcard characters?	false

...

When you ran the previous command to display the help for the `Get-Help` command, you probably noticed the output scrolled by too quickly to read it.

If you're using the PowerShell console, Windows Terminal, or VS Code and need to view a help article, the `help` function can be useful. It pipes the output of `Get-Help` to `more.com`, displaying one page of help content at a time. I recommend using the `help` function instead of the `Get-Help` cmdlet because it provides a better user experience and it's less to type.

ⓘ Note

The ISE doesn't support using `more.com`, so running `help` works the same way as `Get-Help`.

Run each of the following commands in PowerShell on your computer.

PowerShell

```
Get-Help -Name Get-Help -Full  
help -Name Get-Help -Full  
help Get-Help -Full
```

Did you observe any variations in the output when you ran the previous commands?

In the previous example, the first line uses the `Get-Help` cmdlet, the second uses the `help` function, and the third line omits the **Name** parameter while using the `help` function. Since **Name** is a positional parameter, the third example takes advantage of its position instead of explicitly stating the parameter's name.

The difference is that the last two commands display their output one page at a time. When using the `help` function, press the `Spacebar` to display the next page of content or `Q` to quit. If you need to terminate any command running interactively in PowerShell, press `Ctrl + C`.

To quickly find information about a specific parameter, use the **Parameter** parameter. This approach returns content containing only the parameter-specific information, rather than the entire help article. This is the easiest way to find information about a specific parameter.

The following example uses the `help` function with the **Parameter** parameter to return information from the help article for the **Name** parameter of `Get-Help`.

PowerShell

```
help Get-Help -Parameter Name
```

The help information shows that the **Name** parameter is positional and must be specified in the first position (position zero) when used positionally.

Output

```
-Name <System.String>  
Gets help about the specified command or concept. Enter the name of a cmdlet, function, provider, script, or workflow, such as `Get-Member`, a conceptual article name, such as `about_Objects`, or an alias, such as `ls`. Wildcard characters are permitted in cmdlet and provider names, but you can't use wildcard characters to find the names of function help and script help articles.
```

To get help for a script that isn't located in a path that's listed in

the `\$env:Path` environment variable, type the script's path and file name.

If you enter the exact name of a help article, `Get-Help` displays the article contents.

If you enter a word or word pattern that appears in several help article titles, `Get-Help` displays a list of the matching titles.

If you enter any text that doesn't match any help article titles, `Get-Help` displays a list of articles that include that text in their contents.

The names of conceptual articles, such as `about_Objects`, must be entered in English, even in non-English versions of PowerShell.

Required?	false
Position?	0
Default value	None
Accept pipeline input?	True (ByPropertyName)
Accept wildcard characters?	true

The **Name** parameter expects a string value as identified by the `<String>` datatype next to the parameter name.

There are several other parameters you can specify with `Get-Help` to return a subset of a help article. To see how they work, run the following commands on your computer.

PowerShell

```
Get-Help -Name Get-Command -Full
Get-Help -Name Get-Command -Detailed
Get-Help -Name Get-Command -Examples
Get-Help -Name Get-Command -Online
Get-Help -Name Get-Command -Parameter Noun
Get-Help -Name Get-Command -ShowWindow
```

I typically use `help <command name>` with the **Full** or **Online** parameter. If you only have an interest in the examples, use the **Examples** parameter. If you only have an interest in a specific parameter, use the **Parameter** parameter.

When you use the **ShowWindow** parameter, it displays the help content in a separate searchable window. You can move that window to a different monitor if you have multiple monitors. However, the **ShowWindow** parameter has a known bug that might prevent it from displaying the entire help article. The **ShowWindow** parameter also requires an operating system with a Graphical User Interface (GUI). It returns an error when you attempt to use it on Windows Server Core.

If you have internet access, you can use the **Online** parameter instead. The **Online** parameter opens the help article in your default web browser. The online content is the most up-to-date content. The browser allows you to search the help content and view other related help articles.

ⓘ Note

The **Online** parameter isn't supported for **About** articles.

PowerShell

```
help Get-Command -Online
```

Finding commands with Get-Help

To find commands with `Get-Help`, specify a search term surrounded by asterisk (*) wildcard characters for the value of the **Name** parameter. The following example uses the **Name** parameter positionally.

PowerShell

```
help *process*
```

Output

Name	Category	Module	Synops
Enter-PSHostProcess	Cmdlet	Microsoft.PowerShell.Core	Con...
Exit-PSHostProcess	Cmdlet	Microsoft.PowerShell.Core	Clo...
Get-PSHostProcessInfo	Cmdlet	Microsoft.PowerShell.Core	Get...
Debug-Process	Cmdlet	Microsoft.PowerShell.M...	Deb...
Get-Process	Cmdlet	Microsoft.PowerShell.M...	Get...
Start-Process	Cmdlet	Microsoft.PowerShell.M...	Sta...
Stop-Process	Cmdlet	Microsoft.PowerShell.M...	Sto...
Wait-Process	Cmdlet	Microsoft.PowerShell.M...	Wai...
Invoke-LapsPolicyProcessing	Cmdlet	LAPS	Inv...
ConvertTo-ProcessMitigationPolicy	Cmdlet	ProcessMitigations	Con...
Get-ProcessMitigation	Cmdlet	ProcessMitigations	Get...
Set-ProcessMitigation	Cmdlet	ProcessMitigations	Set...

In this scenario, you aren't required to add the (*) wildcard characters. If `Get-Help` can't find a command matching the value you provided, it does a full-text search for that

value. The following example produces the same results as specifying the `*` wildcard character on each end of `process`.

```
PowerShell
```

```
help process
```

When you specify a wildcard character within the value, `Get-Help` only searches for commands that match the pattern you provided. It doesn't perform a full-text search. The following command doesn't return any results.

```
PowerShell
```

```
help pr*cess
```

PowerShell generates an error if you specify a value that begins with a dash without enclosing it in quotes because it interprets it as a parameter name. No such parameter name exists for the `Get-Help` cmdlet.

```
PowerShell
```

```
help -process
```

If you're attempting to search for commands that end with `-process`, you must add an `*` to the beginning of the value.

```
PowerShell
```

```
help *-process
```

When you search for PowerShell commands with `Get-Help`, it's better to be vague rather than too specific.

When you searched for `process` earlier, the results only returned commands that included `process` in their name. But if you search for `processes`, it doesn't find any matches for command names. As previously stated, when help doesn't find any matches, it performs a comprehensive full-text search of every help article on your system and returns those results. This type of search often produces more results than expected, including information not relevant to you.

```
PowerShell
```

help processes

Output

Name	Category	Module	Synops
Disconnect-PSSession	Cmdlet	Microsoft.PowerShell.Core	Dis...
Enter-PSHostProcess	Cmdlet	Microsoft.PowerShell.Core	Con...
ForEach-Object	Cmdlet	Microsoft.PowerShell.Core	Per...
Get-PSHostProcessInfo	Cmdlet	Microsoft.PowerShell.Core	Get...
Get-PSSessionConfiguration	Cmdlet	Microsoft.PowerShell.Core	Get...
New-PSSessionOption	Cmdlet	Microsoft.PowerShell.Core	Cre...
New-PSTransportOption	Cmdlet	Microsoft.PowerShell.Core	Cre...
Out-Host	Cmdlet	Microsoft.PowerShell.Core	Sen...
Start-Job	Cmdlet	Microsoft.PowerShell.Core	Sta...
Where-Object	Cmdlet	Microsoft.PowerShell.Core	Sel...
Debug-Process	Cmdlet	Microsoft.PowerShell.M...	Deb...
Get-Process	Cmdlet	Microsoft.PowerShell.M...	Get...
Get-WmiObject	Cmdlet	Microsoft.PowerShell.M...	Get...
Start-Process	Cmdlet	Microsoft.PowerShell.M...	Sta...
Stop-Process	Cmdlet	Microsoft.PowerShell.M...	Sto...
Wait-Process	Cmdlet	Microsoft.PowerShell.M...	Wai...
Clear-Variable	Cmdlet	Microsoft.PowerShell.U...	Del...
Convert-String	Cmdlet	Microsoft.PowerShell.U...	For...
ConvertFrom-Csv	Cmdlet	Microsoft.PowerShell.U...	Con...
ConvertFrom-Json	Cmdlet	Microsoft.PowerShell.U...	Con...
ConvertTo-Html	Cmdlet	Microsoft.PowerShell.U...	Con...
ConvertTo-Xml	Cmdlet	Microsoft.PowerShell.U...	Cre...
Debug-Runspace	Cmdlet	Microsoft.PowerShell.U...	Sta...
Export-Csv	Cmdlet	Microsoft.PowerShell.U...	Con...
Export-FormatData	Cmdlet	Microsoft.PowerShell.U...	Sav...
Format-List	Cmdlet	Microsoft.PowerShell.U...	For...
Format-Table	Cmdlet	Microsoft.PowerShell.U...	For...
Get-Unique	Cmdlet	Microsoft.PowerShell.U...	Ret...
Group-Object	Cmdlet	Microsoft.PowerShell.U...	Gro...
Import-Clixml	Cmdlet	Microsoft.PowerShell.U...	Imp...
Import-Csv	Cmdlet	Microsoft.PowerShell.U...	Cre...
Measure-Object	Cmdlet	Microsoft.PowerShell.U...	Cal...
Out-File	Cmdlet	Microsoft.PowerShell.U...	Sen...
Out-GridView	Cmdlet	Microsoft.PowerShell.U...	Sen...
Select-Object	Cmdlet	Microsoft.PowerShell.U...	Sel...
Set-Variable	Cmdlet	Microsoft.PowerShell.U...	Set...
Sort-Object	Cmdlet	Microsoft.PowerShell.U...	Sor...
Tee-Object	Cmdlet	Microsoft.PowerShell.U...	Sav...
Trace-Command	Cmdlet	Microsoft.PowerShell.U...	Con...
Write-Information	Cmdlet	Microsoft.PowerShell.U...	Spe...
Export-BinaryMiLog	Cmdlet	CimCmdlets	Cre...
Get-CimAssociatedInstance	Cmdlet	CimCmdlets	Ret...
Get-CimInstance	Cmdlet	CimCmdlets	Get...
Import-BinaryMiLog	Cmdlet	CimCmdlets	Use...
Invoke-CimMethod	Cmdlet	CimCmdlets	Inv...
New-CimInstance	Cmdlet	CimCmdlets	Cre...

Remove-CimInstance	Cmdlet	CimCmdlets	Rem...
Set-CimInstance	Cmdlet	CimCmdlets	Mod...
Compress-Archive	Function	Microsoft.PowerShell.A...	Cre...
Get-Counter	Cmdlet	Microsoft.PowerShell.D...	Get...
Invoke-WSManAction	Cmdlet	Microsoft.WSMan.Manage...	Inv...
Remove-WSManInstance	Cmdlet	Microsoft.WSMan.Manage...	Del...
Get-WSManInstance	Cmdlet	Microsoft.WSMan.Manage...	Dis...
New-WSManInstance	Cmdlet	Microsoft.WSMan.Manage...	Cre...
Set-WSManInstance	Cmdlet	Microsoft.WSMan.Manage...	Mod...
about_Arithmetic_Operators	HelpFile		
about_Arrays	HelpFile		
about_Environment_Variables	HelpFile		
about_Execution_Policies	HelpFile		
about_Functions	HelpFile		
about_Jobs	HelpFile		
aboutLogging	HelpFile		
about_Methods	HelpFile		
about_Objects	HelpFile		
about_Pipelines	HelpFile		
about_Preference_Variables	HelpFile		
about_Remote	HelpFile		
about_Remote_Jobs	HelpFile		
about_Session_Configuration_Files	HelpFile		
about_Simplified_Syntax	HelpFile		
about_Switch	HelpFile		
about_Variables	HelpFile		
about_Variant_Provider	HelpFile		
about_Windows_PowerShell_5.1	HelpFile		
about_WQL	HelpFile		
about_WS-Management_Cmdlets	HelpFile		
about_Foreach-Parallel	HelpFile		
about_Parallel	HelpFile		
about_Sequence	HelpFile		

When you searched for `process`, it returned 12 results. However, when searching for `processes`, it produced 78 results. If your search only finds one match, `Get-Help` displays the help content instead of listing the search results.

```
PowerShell
help *hotfix*

```

Output

```
NAME
    Get-HotFix

SYNOPSIS
    Gets the hotfixes that are installed on local or remote computers.
```

SYNTAX

```
Get-HotFix [-ComputerName <System.String[]>] [-Credential <System.Management.Automation.PSCredential>] [-Description <System.String[]>] [<CommonParameters>]

Get-HotFix [[-Id] <System.String[]>] [-ComputerName <System.String[]>] [-Credential <System.Management.Automation.PSCredential>] [<CommonParameters>]
```

DESCRIPTION

> This cmdlet is only available on the Windows platform. The `Get-HotFix` cmdlet uses the Win32_QuickFixEngineering WMI class to list hotfixes that are installed on the local computer or specified remote computers.

RELATED LINKS

Online Version: https://learn.microsoft.com/powershell/module/microsoft.powershell.management/get-hotfix?view=powershell-5.1&WT.mc_id=ps-gethelp
about_Arrays
Add-Content
Get-ComputerRestorePoint
Get-Credential
Win32_QuickFixEngineering class

REMARKS

To see the examples, type: "Get-Help Get-HotFix -Examples".
For more information, type: "Get-Help Get-HotFix -Detailed".
For technical information, type: "Get-Help Get-HotFix -Full".
For online help, type: "Get-Help Get-HotFix -Online"

You can also find commands that lack help articles with `Get-Help`, although this capability isn't commonly known. The `more` function is one of the commands that doesn't have a help article. To confirm that you can find commands with `Get-Help` that don't include help articles, use the `help` function to find `more`.

PowerShell

```
help *more*
```

The search only found one match, so it returned the basic syntax information you see when a command doesn't have a help article.

Output

NAME

more

SYNTAX

```
more [[-paths] <string[]>]
```

ALIASES

None

REMARKS

None

The PowerShell help system also contains conceptual **About** help articles. You must update the help content on your system to get the **About** articles. For more information, see the [Updating help](#) section of this chapter.

Use the following command to return a list of all **About** help articles on your system.

```
PowerShell
```

```
help About_*
```

When you limit the results to one **About** help article, `Get-Help` displays the content of that article.

```
PowerShell
```

```
help about_Updatable_Help
```

Updating help

Earlier in this chapter, you updated the PowerShell help articles on your computer the first time you ran the `Get-Help` cmdlet. You should periodically run the `Update-Help` cmdlet on your computer to obtain any updates to the help content.

 **Important**

In Windows PowerShell 5.1, you must run `Update-Help` as an administrator in an elevated PowerShell session.

In the following example, `Update-Help` downloads the PowerShell help content for all modules installed on your computer. You should use the **Force** parameter to ensure that you download the latest version of the help content.

```
PowerShell
```

```
Update-Help -Force
```

As shown in the following results, a module returned an error. Errors aren't uncommon and usually occur when the module's author doesn't configure updatable help correctly.

Output

```
Update-Help : Failed to update Help for the module(s) 'BitsTransfer' with UI culture(s) {en-US} : Unable to retrieve the HelpInfo XML file for UI culture en-US. Make sure the HelpInfoUri property in the module manifest is valid or check your network connection and then try the command again.  
At line:1 char:1  
+ Update-Help  
+ ~~~~~~  
    + CategoryInfo          : ResourceUnavailable: (:) [Update-Help], Except  
      ion  
    + FullyQualifiedErrorId : UnableToRetrieveHelpInfoXml,Microsoft.PowerShe  
      ll.Commands.UpdateHelpCommand
```

`Update-Help` requires internet access to download the help content. If your computer doesn't have internet access, use the `Save-Help` cmdlet on a computer with internet access to download and save the updated help content. Then, use the **SourcePath** parameter of `Update-Help` to specify the location of the saved updated help content.

Get-Command

`Get-Command` is another multipurpose command that helps you find commands. When you run `Get-Command` without any parameters, it returns a list of all PowerShell commands on your system. You can also use `Get-Command` to get command syntax similar to `Get-Help`.

How do you determine the syntax for `Get-Command`? You could use `Get-Help` to display the help article for `Get-Command`, as shown in the [Get-Help](#) section of this chapter. You can also use `Get-Command` with the **Syntax** parameter to view the syntax for any command. This shortcut helps you quickly determine how to use a command without navigating through its help content.

PowerShell

```
Get-Command -Name Get-Command -Syntax
```

Using `Get-Command` with the `Syntax` parameter provides a more concise view of the syntax that shows the parameters and their value types, without listing the specific allowable values like `Get-Help` shows.

Output

```
Get-Command [[-ArgumentList] <Object[]>] [-Verb <string[]>]
[-Noun <string[]>] [-Module <string[]>]
[-FullyQualifiedModule <ModuleSpecification[]>] [-TotalCount <int>]
[-Syntax] [-ShowCommandInfo] [-All] [-ListImported]
[-ParameterName <string[]>] [-ParameterType <PSTypeName[]>]
[<CommonParameters>]

Get-Command [[-Name] <string[]>] [[-ArgumentList] <Object[]>]
[-Module <string[]>] [-FullyQualifiedModule <ModuleSpecification[]>]
[- CommandType <CommandTypes>] [-TotalCount <int>] [-Syntax]
[-ShowCommandInfo] [-All] [-ListImported] [-ParameterName <string[]>]
[-ParameterType <PSTypeName[]>] [<CommonParameters>]
```

If you need more detailed information about how to use a command, use `Get-Help`.

PowerShell

```
help Get-Command -Full
```

The **SYNTAX** section of `Get-Help` provides a more user-friendly display by expanding enumerated values for parameters. It shows you the actual values you can use, making it easier to understand the available options.

Output

```
...
Get-Command [[-Name] <System.String[]>] [[-ArgumentList]
<System.Object[]>] [-All] [-CommandType {Alias | Function | Filter |
Cmdlet | ExternalScript | Application | Script | Workflow |
Configuration | All}] [-FullyQualifiedModule
<Microsoft.PowerShell.Commands.ModuleSpecification[]>] [-ListImported]
[-Module <System.String[]>] [-ParameterName <System.String[]>]
[-ParameterType <System.Management.Automation.PSTypeName[]>]
[-ShowCommandInfo] [-Syntax] [-TotalCount <System.Int32>]
[<CommonParameters>]

Get-Command [[-ArgumentList] <System.Object[]>] [-All]
[-FullyQualifiedModule
<Microsoft.PowerShell.Commands.ModuleSpecification[]>] [-ListImported]
[-Module <System.String[]>] [-Noun <System.String[]>] [-ParameterName
<System.String[]>] [-ParameterType
<System.Management.Automation.PSTypeName[]>] [-ShowCommandInfo]
[-Syntax] [-TotalCount <System.Int32>] [-Verb <System.String[]>]
```

```
[<CommonParameters>]
```

```
...
```

The **PARAMETERS** section of the help for `Get-Command` reveals that the **Name**, **Noun**, and **Verb** parameters accept wildcard characters.

Output

```
...
```

```
-Name <System.String[]>
```

Specifies an array of names. This cmdlet gets only commands that have the specified name. Enter a name or name pattern. Wildcard characters are permitted.

To get commands that have the same name, use the **All** parameter. When two commands have the same name, by default, `Get-Command` gets the command that runs when you type the command name.

Required?	false
-----------	-------

Position?	0
-----------	---

Default value	None
---------------	------

Accept pipeline input?	True (ByPropertyName, ByValue)
------------------------	--------------------------------

Accept wildcard characters?	true
-----------------------------	------

```
-Noun <System.String[]>
```

Specifies an array of command nouns. This cmdlet gets commands, which include cmdlets, functions, and aliases, that have names that include the specified noun. Enter one or more nouns or noun patterns. Wildcard characters are permitted.

Required?	false
-----------	-------

Position?	named
-----------	-------

Default value	None
---------------	------

Accept pipeline input?	True (ByPropertyName)
------------------------	-----------------------

Accept wildcard characters?	true
-----------------------------	------

```
-Verb <System.String[]>
```

Specifies an array of command verbs. This cmdlet gets commands, which include cmdlets, functions, and aliases, that have names that include the specified verb. Enter one or more verbs or verb patterns. Wildcard characters are permitted.

Required?	false
-----------	-------

Position?	named
-----------	-------

Default value	None
---------------	------

Accept pipeline input?	True (ByPropertyName)
------------------------	-----------------------

Accept wildcard characters?	true
-----------------------------	------

```
...
```

The following example uses the `*` wildcard character with the value for the **Name** parameter of `Get-Command`.

```
PowerShell
```

```
Get-Command -Name *service*
```

When you use wildcard characters with the **Name** parameter of `Get-Command`, it returns PowerShell commands and native commands, as shown in the following results.

Output

CommandType	Name	Version
Function	Get-NetFirewallServiceFilter	2.0.0.0
Function	Set-NetFirewallServiceFilter	2.0.0.0
Cmdlet	Get-Service	3.1.0.0
Cmdlet	New-Service	3.1.0.0
Cmdlet	New-WebServiceProxy	3.1.0.0
Cmdlet	Restart-Service	3.1.0.0
Cmdlet	Resume-Service	3.1.0.0
Cmdlet	Set-Service	3.1.0.0
Cmdlet	Start-Service	3.1.0.0
Cmdlet	Stop-Service	3.1.0.0
Cmdlet	Suspend-Service	3.1.0.0
Application	SecurityHealthService.exe	10.0.2...
Application	SensorDataService.exe	10.0.2...
Application	services.exe	10.0.2...
Application	services.msc	0.0.0.0
Application	TieringEngineService.exe	10.0.2...
Application	Windows.WARP.JITService.exe	10.0.2...

You can limit the results of `Get-Command` to PowerShell commands using the **CommandType** parameter.

```
PowerShell
```

```
Get-Command -Name *service* -CommandType Cmdlet, Function, Alias, Script
```

Another option might be to use either the **Verb** or **Noun** parameter or both since only PowerShell commands have verbs and nouns.

The following example uses `Get-Command` to find commands on your computer that work with processes. Use the **Noun** parameter and specify `Process` as its value.

```
PowerShell
```

```
Get-Command -Noun Process
```

Output		
CommandType	Name	Version
Cmdlet	Debug-Process	3.1.0.0
Cmdlet	Get-Process	3.1.0.0
Cmdlet	Start-Process	3.1.0.0
Cmdlet	Stop-Process	3.1.0.0
Cmdlet	Wait-Process	3.1.0.0

Summary

In this chapter, you learned how to find commands with `Get-Help` and `Get-Command`. You also learned how to use the help system to understand how to use commands once you find them. In addition, you learned how to update the help system on your computer when new help content is available.

Review

1. Is the `DisplayName` parameter of `Get-Service` positional?
2. How many parameter sets does the `Get-Process` cmdlet have?
3. What PowerShell commands exist for working with event logs?
4. What's the PowerShell command for returning a list of PowerShell processes running on your computer?
5. How do you update the PowerShell help content stored on your computer?

References

To learn more about the concepts covered in this chapter, read the following PowerShell help articles.

- [Get-Help](#)
- [Get-Command](#)
- [Update-Help](#)
- [Save-Help](#)
- [about_Updateable_Help](#)
- [about_Command_Syntax](#)

Next steps

In the next chapter, you'll learn about objects, properties, methods, and the `Get-Member` cmdlet.

Chapter 3 - Discovering objects, properties, and methods

Article • 08/19/2024

PowerShell is an object-oriented scripting language. It represents data and system states using structured objects derived from .NET classes defined in the .NET Framework. By leveraging the .NET Framework, PowerShell offers access to various system capabilities, including file system, registry, and Windows Management Instrumentation (WMI) classes. PowerShell also has access to the .NET Framework class library, which contains a vast collection of classes that you can use to develop robust PowerShell scripts.

In PowerShell, each item or state is an instance of an object that can be explored and manipulated. The `Get-Member` cmdlet is one of the primary tools provided by PowerShell for object discovery, which reveals an object's characteristics. This chapter explores how PowerShell leverages objects and how you can discover and manipulate these objects to streamline your scripts and manage your systems more efficiently.

Prerequisites

To follow the specific examples in this chapter, ensure that your lab environment computer is part of your lab environment Active Directory domain. You must also install the Active Directory PowerShell module bundled with the Windows Remote Server Administration Tools (RSAT). If you're using Windows 10 build 1809 or later, including Windows 11, you can install RSAT as a Windows feature.

Note

Active Directory is unsupported for Windows Home editions.

- For information about installing RSAT, see [Windows Management modules](#).
- For older versions of Windows, see [RSAT for Windows](#).

Get-Member

`Get-Member` provides insight into the objects, properties, and methods associated with PowerShell commands. You can pipe any PowerShell command that produces object-based output to `Get-Member`. When you pipe the output of a command to `Get-Member`, it

reveals the structure of the object returned by the command, detailing its properties and methods.

- **Properties:** The attributes of an object.
- **Methods:** The actions you can perform on an object.

To illustrate this concept, consider a driver's license as an analogy. Like any object, a driver's license has properties, such as **eye color**, which typically includes `blue` and `brown` values. In contrast, methods represent actions you can perform on the object. For instance, **Revoke** is a method that the Department of Motor Vehicles can perform on a driver's license.

Properties

To retrieve details about the Windows Time service on your system using PowerShell, use the `Get-Service` cmdlet.

```
PowerShell
```

```
Get-Service -Name w32time
```

The results include the **Status**, **Name**, and **DisplayName** properties. The **Status** property indicates that the service is `Running`. The value for the **Name** property is `w32time`, and the value for the **DisplayName** property is `Windows Time`.

```
Output
```

Status	Name	DisplayName
-----	----	-----
Running	w32time	Windows Time

To list all available properties and methods for `Get-Service`, pipe it to `Get-Member`.

```
PowerShell
```

```
Get-Service -Name w32time | Get-Member
```

The results show the first line contains one piece of significant information. **TypeName** identifies the type of object returned, which in this example is a **System.ServiceProcess.ServiceController** object. This name is often abbreviated to the last part of the **TypeName**, such as **ServiceController**, in this example.

```
Output
```

TypeName: System.ServiceProcess.ServiceController

Name	MemberType	Definition
Name	AliasProperty	Name = ServiceName
RequiredServices	AliasProperty	RequiredServices = ServicesDepend...
Disposed	Event	System.EventHandler Disposed(Syst...
Close	Method	void Close()
Continue	Method	void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef Cr...
Dispose	Method	void Dispose(), void IDisposable....
Equals	Method	bool Equals(System.Object obj)
ExecuteCommand	Method	void ExecuteCommand(int command)
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeService()
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object InitializeLifetimeS...
Pause	Method	void Pause()
Refresh	Method	void Refresh()
Start	Method	void Start(), void Start(string[]...)
Stop	Method	void Stop()
WaitForStatus	Method	void WaitForStatus(System.Service...
CanPauseAndContinue	Property	bool CanPauseAndContinue {get;}
CanShutdown	Property	bool CanShutdown {get;}
CanStop	Property	bool CanStop {get;}
Container	Property	System.ComponentModel.IContainer ...
DependentServices	Property	System.ServiceProcess.ServiceCont...
DisplayName	Property	string DisplayName {get;set;}
MachineName	Property	string MachineName {get;set;}
ServiceHandle	Property	System.Runtime.InteropServices.Sa...
ServiceName	Property	string ServiceName {get;set;}
ServicesDependedOn	Property	System.ServiceProcess.ServiceCont...
ServiceType	Property	System.ServiceProcess.ServiceType...
Site	Property	System.ComponentModel.ISite Site ...
StartType	Property	System.ServiceProcess.ServiceStar...
Status	Property	System.ServiceProcess.ServiceCont...
ToString	ScriptMethod	System.Object ToString();

Notice when you piped `Get-Service` to `Get-Member`, there are more properties than are displayed by default. Although these additional properties aren't shown by default, you can select them by piping to `Select-Object` and using the `Property` parameter. The following example selects all properties by piping the results of `Get-Service` to `Select-Object` and specifying the `*` wildcard character as the value for the `Property` parameter.

PowerShell

```
Get-Service -Name w32time | Select-Object -Property *
```

By default, PowerShell returns four properties as a table and five or more as a list. However, some commands apply custom formatting to override the default number of properties displayed in a table. You can use `Format-Table` and `Format-List` to override these defaults manually.

Output

```
Name          : w32time
RequiredServices : {}
CanPauseAndContinue : False
CanShutdown      : True
CanStop         : True
DisplayName     : Windows Time
DependentServices : {}
MachineName    : .
ServiceName    : w32time
ServicesDependedOn : {}
ServiceHandle   :
Status         : Running
ServiceType    : Win32OwnProcess, Win32ShareProcess
StartType       : Manual
Site           :
Container      :
```

Specific properties can also be selected using a comma-separated list as the value of the **Property** parameter.

PowerShell

```
Get-Service -Name w32time |
  Select-Object -Property Status, Name, DisplayName, ServiceType
```

Output

Status	Name	DisplayName	ServiceType
Running	w32time	Windows Time	Win32OwnProcess, Win32ShareProcess

You can use wildcard characters when specifying property names with `Select-Object`.

In the following example, use `Can*` as one of the values for the **Property** parameter to return all the properties that start with `Can`. These include `CanPauseAndContinue`, `CanShutdown`, and `CanStop`.

PowerShell

```
Get-Service -Name w32time |  
    Select-Object -Property Status, DisplayName, Can*
```

Notice there are more properties listed than are displayed by default.

Output

```
Status          : Running  
DisplayName     : Windows Time  
CanPauseAndContinue : False  
CanShutdown     : True  
CanStop         : True
```

Methods

Methods are actions you can perform on an object. Use the **MemberType** parameter to narrow down the results of `Get-Member` to display only the methods for `Get-Service`.

PowerShell

```
Get-Service -Name w32time | Get-Member -MemberType Method
```

As you can see, there are several methods.

Output

```
TypeName: System.ServiceProcess.ServiceController  
  
Name           MemberType Definition  
---  
Close          Method    void Close()  
Continue       Method    void Continue()  
CreateObjRef   Method    System.Runtime.Remoting.ObjRef Creat...  
Dispose        Method    void Dispose(), void IDisposable.Dis...  
Equals         Method    bool Equals(System.Object obj)  
ExecuteCommand Method    void ExecuteCommand(int command)  
GetHashCode    Method    int GetHashCode()  
GetLifetimeService Method  System.Object GetLifetimeService()  
GetType        Method    type GetType()  
InitializeLifetimeService Method System.Object InitializeLifetimeServ...  
Pause          Method    void Pause()  
Refresh        Method    void Refresh()  
Start          Method    void Start(), void Start(string[] args)  
Stop           Method    void Stop()  
WaitForStatus  Method    void WaitForStatus(System.ServicePro...
```

You can use the **Stop** method to stop a Windows service. You must run this command from an elevated PowerShell session.

```
PowerShell
```

```
(Get-Service -Name w32time).Stop()
```

Query the status of the Windows Time service to confirm it's stopped.

```
PowerShell
```

```
Get-Service -Name w32time
```

```
Output
```

Status	Name	DisplayName
-----	-----	-----
Stopped	w32time	Windows Time

You might use methods sparingly, but you should be aware of them. Sometimes, you find `Get-*` commands without a corresponding `Set-*` command. Often, you can find a method to perform a `Set-*` action in this scenario. The `Get-SqlAgentJob` cmdlet in the **SqlServer** PowerShell module is an excellent example. No corresponding `Set-*` cmdlet exists, but you can use a method to complete the same task. For more information about the **SqlServer** PowerShell module and installation instructions, see the [SQL Server PowerShell overview](#).

Another reason to be aware of methods is some PowerShell users assume you can't make destructive changes with `Get-*` commands, but they can actually cause severe problems if misused.

A better option is to use a dedicated cmdlet if one exists to perform an action. For example, use the `Start-Service` cmdlet to start the Windows Time service.

By default, `Start-Service`, like the **Start** method of `Get-Service`, doesn't return any results. However, one of the benefits of using a cmdlet is that it often provides additional capabilities that aren't available with a method.

In the following example, use the **PassThru** parameter, which causes a cmdlet that doesn't typically produce output to generate output.

Since PowerShell doesn't participate in User Access Control (UAC), you must run commands that require elevation, such as `Start-Service`, from an elevated PowerShell

session.

PowerShell

```
Get-Service -Name w32time | Start-Service -PassThru
```

Output

Status	Name	DisplayName
-----	-----	-----
Running	w32time	Windows Time

ⓘ Note

When working with PowerShell cmdlets, it's important to avoid making assumptions about their output.

To retrieve information about the PowerShell process running on your lab environment computer, use the `Get-Process` cmdlet.

PowerShell

```
Get-Process -Name powershell
```

Output

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	-----	--	-	-----
710	31	55692	70580	0.72	9436	2	powershell

To determine the available properties, pipe `Get-Process` to `Get-Member`.

PowerShell

```
Get-Process -Name powershell | Get-Member
```

When using the `Get-Process` command, you might notice that some properties displayed by default are missing when you view the results of `Get-Member`. This behavior is because several of the values shown by default, such as `NPM(K)`, `PM(K)`, `WS(K)`, and `CPU(s)`, are calculated properties. You must pipe commands to `Get-Member` to determine their actual property names.

Output

TypeName: System.Diagnostics.Process

Name	MemberType	Definition
Handles	AliasProperty	Handles = HandleCount
Name	AliasProperty	Name = ProcessName
NPM	AliasProperty	NPM = NonpagedSystemMemorySize64
PM	AliasProperty	PM = PagedMemorySize64
SI	AliasProperty	SI = SessionId
VM	AliasProperty	VM = VirtualMemorySize64
WS	AliasProperty	WS = WorkingSet64
Disposed	Event	System.EventHandler Disposed(Sy...)
ErrorDataReceived	Event	System.Diagnostics.DataReceived...
Exited	Event	System.EventHandler Exited(Syst...
OutputDataReceived	Event	System.Diagnostics.DataReceived...
BeginErrorReadLine	Method	void BeginErrorReadLine()
BeginOutputReadLine	Method	void BeginOutputReadLine()
CancelErrorRead	Method	void CancelErrorRead()
CancelOutputRead	Method	void CancelOutputRead()
Close	Method	void Close()
CloseMainWindow	Method	bool CloseMainWindow()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef ...
Dispose	Method	void Dispose(), void IDisposable...
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeService()
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object InitializeLifetim...
Kill	Method	void Kill()
Refresh	Method	void Refresh()
Start	Method	bool Start()
ToString	Method	string ToString()
WaitForExit	Method	bool WaitForExit(int millisecond...
WaitForInputIdle	Method	bool WaitForInputIdle(int milli...
__NounName	NoteProperty	string __NounName=Process
BasePriority	Property	int BasePriority {get;}
Container	Property	System.ComponentModel.IContainer...
EnableRaisingEvents	Property	bool EnableRaisingEvents {get; s...
ExitCode	Property	int ExitCode {get;}
ExitTime	Property	datetime ExitTime {get;}
Handle	Property	System.IntPtr Handle {get;}
HandleCount	Property	int HandleCount {get;}
HasExited	Property	bool HasExited {get;}
Id	Property	int Id {get;}
MachineName	Property	string MachineName {get;}
MainModule	Property	System.Diagnostics.ProcessModul...
MainWindowHandle	Property	System.IntPtr MainWindowHandle ...
MainWindowTitle	Property	string MainWindowTitle {get;}
MaxWorkingSet	Property	System.IntPtr MaxWorkingSet {ge...
MinWorkingSet	Property	System.IntPtr MinWorkingSet {ge...
Modules	Property	System.Diagnostics.ProcessModul...
NonpagedSystemMemorySize	Property	int NonpagedSystemMemorySize {g...

NonpagedSystemMemorySize64	Property	long NonpagedSystemMemorySize64...
PagedMemorySize	Property	int PagedMemorySize {get;}
PagedMemorySize64	Property	long PagedMemorySize64 {get;}
PagedSystemMemorySize	Property	int PagedSystemMemorySize {get;}
PagedSystemMemorySize64	Property	long PagedSystemMemorySize64 {g...
PeakPagedMemorySize	Property	int PeakPagedMemorySize {get;}
PeakPagedMemorySize64	Property	long PeakPagedMemorySize64 {get;}
PeakVirtualMemorySize	Property	int PeakVirtualMemorySize {get;}
PeakVirtualMemorySize64	Property	long PeakVirtualMemorySize64 {g...
PeakWorkingSet	Property	int PeakWorkingSet {get;}
PeakWorkingSet64	Property	long PeakWorkingSet64 {get;}
PriorityBoostEnabled	Property	bool PriorityBoostEnabled {get;...}
PriorityClass	Property	System.Diagnostics.ProcessPrior...
PrivateMemorySize	Property	int PrivateMemorySize {get;}
PrivateMemorySize64	Property	long PrivateMemorySize64 {get;}
PrivilegedProcessorTime	Property	timespan PrivilegedProcessorTim...
ProcessName	Property	string ProcessName {get;}
ProcessorAffinity	Property	System.IntPtr ProcessorAffinity...
Responding	Property	bool Responding {get;}
SafeHandle	Property	Microsoft.Win32.SafeHandles.Saf...
SessionId	Property	int SessionId {get;}
Site	Property	System.ComponentModel.ISite Sit...
StandardError	Property	System.IO.StreamReader Standard...
StandardInput	Property	System.IO.StreamWriter Standard...
StandardOutput	Property	System.IO.StreamReader Standard...
StartInfo	Property	System.Diagnostics.ProcessStart...
StartTime	Property	datetime StartTime {get;}
SynchronizingObject	Property	System.ComponentModel.ISynchron...
Threads	Property	System.Diagnostics.ProcessThrea...
TotalProcessorTime	Property	timespan TotalProcessorTime {get;}
UserProcessorTime	Property	timespan UserProcessorTime {get;}
VirtualMemorySize	Property	int VirtualMemorySize {get;}
VirtualMemorySize64	Property	long VirtualMemorySize64 {get;}
WorkingSet	Property	int WorkingSet {get;}
WorkingSet64	Property	long WorkingSet64 {get;}
PSConfiguration	PropertySet	PSConfiguration {Name, Id, Prio...
PSResources	PropertySet	PSResources {Name, Id, Handleco...
Company	ScriptProperty	System.Object Company {get=\$thi...
CPU	ScriptProperty	System.Object CPU {get=\$this.To...
Description	ScriptProperty	System.Object Description {get=...
FileVersion	ScriptProperty	System.Object FileVersion {get=...
Path	ScriptProperty	System.Object Path {get=\$this.M...
Product	ScriptProperty	System.Object Product {get=\$thi...
ProductVersion	ScriptProperty	System.Object ProductVersion {g...

You can't pipe a command to `Get-Member` that doesn't generate output. Because `Start-Service` doesn't produce output by default, attempting to pipe it to `Get-Member` results in an error.

PowerShell

```
Start-Service -Name w32time | Get-Member
```

ⓘ Note

To be piped to `Get-Member`, a command must produce object-based output.

Output

```
Get-Member : You must specify an object for the Get-Member cmdlet.  
At line:1 char:31  
+ Start-Service -Name w32time | Get-Member  
+  
+ CategoryInfo          : CloseError: (:) [Get-Member], InvalidOperationException  
Exception  
+ FullyQualifiedErrorId : NoObjectInGetMember,Microsoft.PowerShell.Commands.GetMemberCommand
```

To avoid this error, specify the `PassThru` parameter with `Start-Service`. As previously mentioned, adding the `PassThru` parameter causes a cmdlet that doesn't usually produce output to generate output.

PowerShell

```
Start-Service -Name w32time -PassThru | Get-Member
```

Output

TypeName: System.ServiceProcess.ServiceController

Name	MemberType	Definition
-----	-----	-----
Name	AliasProperty	Name = ServiceName
RequiredServices	AliasProperty	RequiredServices = ServicesDepend...
Disposed	Event	System.EventHandler Disposed(Syst...
Close	Method	void Close()
Continue	Method	void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef Cr...
Dispose	Method	void Dispose(), void IDisposable....
Equals	Method	bool Equals(System.Object obj)
ExecuteCommand	Method	void ExecuteCommand(int command)
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeService()
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object InitializeLifetimeS...
Pause	Method	void Pause()
Refresh	Method	void Refresh()

Start	Method	void Start(), void Start(string[]...)
Stop	Method	void Stop()
WaitForStatus	Method	void WaitForStatus(System.Service...
CanPauseAndContinue	Property	bool CanPauseAndContinue {get;}
CanShutdown	Property	bool CanShutdown {get;}
CanStop	Property	bool CanStop {get;}
Container	Property	System.ComponentModel.IContainer ...
DependentServices	Property	System.ServiceProcess.ServiceCont...
DisplayName	Property	string DisplayName {get;set;}
MachineName	Property	string MachineName {get;set;}
ServiceHandle	Property	System.Runtime.InteropServices.Sa...
ServiceName	Property	string ServiceName {get;set;}
ServicesDependedOn	Property	System.ServiceProcess.ServiceCont...
ServiceType	Property	System.ServiceProcess.ServiceType...
Site	Property	System.ComponentModel.ISite Site ...
StartType	Property	System.ServiceProcess.ServiceStar...
Status	Property	System.ServiceProcess.ServiceCont...
ToString	ScriptMethod	System.Object ToString();

`Out-Host` is designed to show output directly in the PowerShell host and doesn't produce object-based output. As a result, you can't pipe its output to `Get-Member`, which requires object-based input.

PowerShell

```
Get-Service -Name w32time | Out-Host | Get-Member
```

Output

Status	Name	DisplayName
Running	w32time	Windows Time

```
Get-Member : You must specify an object for the Get-Member cmdlet.
At line:1 char:40
+ Get-Service -Name w32time | Out-Host | Get-Member
+               ~~~~~
+ CategoryInfo          : CloseError: (:) [Get-Member], InvalidOperationException
Exception
+ FullyQualifiedErrorId : NoObjectInGetMember,Microsoft.PowerShell.Commands.GetMemberCommand
```

Get-Command

Knowing the type of object a command produces allows you to search for commands that accept that type of object as input.

```
PowerShell
```

```
Get-Command -ParameterType ServiceController
```

The following commands accept a **ServiceController** object via pipeline or parameter input.

```
Output
```

CommandType	Name	Version
Cmdlet	Get-Service	3.1.0.0
Cmdlet	Restart-Service	3.1.0.0
Cmdlet	Resume-Service	3.1.0.0
Cmdlet	Set-Service	3.1.0.0
Cmdlet	Start-Service	3.1.0.0
Cmdlet	Stop-Service	3.1.0.0
Cmdlet	Suspend-Service	3.1.0.0

Active Directory

ⓘ Note

As mentioned in the chapter prerequisites, ensure you have RSAT installed for this section. Additionally, your lab environment computer must be a member of your lab environment Active Directory domain.

To identify the commands added to the **ActiveDirectory** PowerShell module after you install RSAT, use `Get-Command` combined with the **Module** parameter. The following example lists all the commands available in the **ActiveDirectory** module.

```
PowerShell
```

```
Get-Command -Module ActiveDirectory
```

The **ActiveDirectory** PowerShell module added a total of 147 commands.

Have you observed the naming convention of these commands? The nouns in the command names are prefixed with *AD* to avoid potential naming conflicts with commands in other modules. This prefixing is a common practice among PowerShell modules.

```
Output
```

CommandType	Name	Version
Cmdlet	Add-ADCentralAccessPolicyMember	1.0.1.0
Cmdlet	Add-ADComputerServiceAccount	1.0.1.0
Cmdlet	Add-ADDomainControllerPasswordReplicationPolicy	1.0.1.0
Cmdlet	Add-ADFineGrainedPasswordPolicySubject	1.0.1.0
Cmdlet	Add-ADGroupMember	1.0.1.0
Cmdlet	Add-ADPrincipalGroupMembership	1.0.1.0
Cmdlet	Add-ADResourcePropertyListMember	1.0.1.0
Cmdlet	Clear-ADAccountExpiration	1.0.1.0
Cmdlet	Clear-ADClaimTransformLink	1.0.1.0
Cmdlet	Disable-ADAccount	1.0.1.0
...		

By default, the `Get-ADUser` cmdlet retrieves a limited set of properties for user objects and limits its output to the first 1,000 users. This constraint is a performance optimization designed to avoid overwhelming Active Directory with excessive data retrieval.

PowerShell

```
Get-ADUser -Identity mike | Get-Member -MemberType Properties
```

Even if you only have a basic understanding of Active Directory, you might recognize that a user account has more properties than those shown in the example.

Output

```
TypeName: Microsoft.ActiveDirectory.Management.ADUser
```

Name	MemberType	Definition
DistinguishedName	Property	System.String DistinguishedName {get;set;}
Enabled	Property	System.Boolean Enabled {get;set;}
GivenName	Property	System.String GivenName {get;set;}
Name	Property	System.String Name {get;}
ObjectClass	Property	System.String ObjectClass {get;set;}
ObjectGUID	Property	System.Nullable`1[[System.Guid, mscorelib, Ve...]
SamAccountName	Property	System.String SamAccountName {get;set;}
SID	Property	System.Security.Principal.SecurityIdentifier...
Surname	Property	System.String Surname {get;set;}
UserPrincipalName	Property	System.String UserPrincipalName {get;set;}

The `Get-ADUser` cmdlet includes a `Properties` parameter to specify additional properties beyond the defaults you want to retrieve. To return all properties, use the `*` wildcard character as the parameter value.

PowerShell

```
Get-ADUser -Identity mike -Properties * | Get-Member -MemberType Properties
```

Output

TypeName: Microsoft.ActiveDirectory.Management.ADUser

Name	MemberType	Definition
AccountExpirationDate	Property	System.DateTime AccountEx...
accountExpires	Property	System.Int64 accountExpir...
AccountLockoutTime	Property	System.DateTime AccountLo...
AccountNotDelegated	Property	System.Boolean AccountNot...
AllowReversiblePasswordEncryption	Property	System.Boolean AllowRever...
AuthenticationPolicy	Property	Microsoft.ActiveDirectory...
AuthenticationPolicySilo	Property	Microsoft.ActiveDirectory...
BadLogonCount	Property	System.Int32 BadLogonCoun...
badPasswordTime	Property	System.Int64 badPasswordT...
badPwdCount	Property	System.Int32 badPwdCount ...
CannotChangePassword	Property	System.Boolean CannotChan...
CanonicalName	Property	System.String CanonicalNa...
Certificates	Property	Microsoft.ActiveDirectory...
City	Property	System.String City {get;s...
CN	Property	System.String CN {get;}
codePage	Property	System.Int32 codePage {ge...
Company	Property	System.String Company {ge...
CompoundIdentitySupported	Property	Microsoft.ActiveDirectory...
Country	Property	System.String Country {ge...
countryCode	Property	System.Int32 countryCode ...
Created	Property	System.DateTime Created {...
createTimeStamp	Property	System.DateTime createTime...
Deleted	Property	System.Boolean Deleted {g...
Department	Property	System.String Department ...
Description	Property	System.String Description...
DisplayName	Property	System.String DisplayName...
DistinguishedName	Property	System.String Distinguish...
Division	Property	System.String Division {g...
DoesNotRequirePreAuth	Property	System.Boolean DoesNotReq...
dSCorePropagationData	Property	Microsoft.ActiveDirectory...
EmailAddress	Property	System.String EmailAddres...
EmployeeID	Property	System.String EmployeeID ...
EmployeeNumber	Property	System.String EmployeeNum...
Enabled	Property	System.Boolean Enabled {g...
Fax	Property	System.String Fax {get;set;}
GivenName	Property	System.String GivenName {...
HomeDirectory	Property	System.String HomeDirecto...
HomedirRequired	Property	System.Boolean HomedirReq...
HomeDrive	Property	System.String HomeDrive {...
HomePage	Property	System.String HomePage {g...
HomePhone	Property	System.String HomePhone {...
Initials	Property	System.String Initials {g...
instanceType	Property	System.Int32 instanceType...

isDeleted	Property	System.Boolean isDeleted ...
KerberosEncryptionType	Property	Microsoft.ActiveDirectory... System.String KerberosEncr...
LastBadPasswordAttempt	Property	System.DateTime LastBadPa...
LastKnownParent	Property	System.String LastKnownPa...
lastLogoff	Property	System.Int64 lastLogoff {...}
lastLogon	Property	System.Int64 lastLogon {g...
LastLogonDate	Property	System.DateTime LastLogon...
lastLogonTimestamp	Property	System.Int64 lastLogonTim...
LockedOut	Property	System.Boolean LockedOut ...
logonCount	Property	System.Int32 logonCount {...
LogonWorkstations	Property	System.String LogonWorkst...
Manager	Property	System.String Manager {ge...
MemberOf	Property	Microsoft.ActiveDirectory...
MNSLogonAccount	Property	System.Boolean MNSLogonAc...
MobilePhone	Property	System.String MobilePhone...
Modified	Property	System.DateTime Modified ...
modifyTimeStamp	Property	System.DateTime modifyTim...
msDS-User-Account-Control-Computed	Property	System.Int32 msDS-User-Ac...
Name	Property	System.String Name {get;}
nTSecurityDescriptor	Property	System.DirectoryServices...
ObjectCategory	Property	System.String ObjectCateg...
ObjectClass	Property	System.String ObjectClass...
ObjectGUID	Property	System.Nullable`1[[System...
objectSid	Property	System.Security.Principal...
Office	Property	System.String Office {get...
OfficePhone	Property	System.String OfficePhone...
Organization	Property	System.String Organizatio...
OtherName	Property	System.String OtherName {...
PasswordExpired	Property	System.Boolean PasswordEx...
PasswordLastSet	Property	System.DateTime PasswordL...
PasswordNeverExpires	Property	System.Boolean PasswordNe...
PasswordNotRequired	Property	System.Boolean PasswordNo...
POBox	Property	System.String POBox {get;...}
PostalCode	Property	System.String PostalCode ...
PrimaryGroup	Property	System.String PrimaryGrou...
primaryGroupID	Property	System.Int32 primaryGroup...
PrincipalsAllowedToDelegateToAccount	Property	Microsoft.ActiveDirectory...
ProfilePath	Property	System.String ProfilePath...
ProtectedFromAccidentalDeletion	Property	System.Boolean ProtectedF...
pwdLastSet	Property	System.Int64 pwdLastSet {...
SamAccountName	Property	System.String SamAccountN...
sAMAccountType	Property	System.Int32 sAMAccountTy...
ScriptPath	Property	System.String ScriptPath ...
sDRightsEffective	Property	System.Int32 sDRightsEffe...
ServicePrincipalNames	Property	Microsoft.ActiveDirectory...
SID	Property	System.Security.Principal...
SIDHistory	Property	Microsoft.ActiveDirectory...
SmartcardLogonRequired	Property	System.Boolean SmartcardL...
sn	Property	System.String sn {get;set;}
State	Property	System.String State {get;...}
StreetAddress	Property	System.String StreetAddre...
Surname	Property	System.String Surname {ge...
Title	Property	System.String Title {get;...}
TrustedForDelegation	Property	System.Boolean TrustedFor...
TrustedToAuthForDelegation	Property	System.Boolean TrustedToA...

UseDESKeyOnly	Property	System.Boolean UseDESKeyO...
userAccountControl	Property	System.Int32 userAccountC...
userCertificate	Property	Microsoft.ActiveDirectory...
UserPrincipalName	Property	System.String UserPrincip...
uSNChanged	Property	System.Int64 uSNChanged {...
uSNCreated	Property	System.Int64 uSNCreated {...
whenChanged	Property	System.DateTime whenChang...
whenCreated	Property	System.DateTime whenCreat...

The default configuration for retrieving Active Directory user account properties is intentionally limited to avoid performance issues. Trying to return every property for every user account in your production Active Directory environment could severely degrade the performance of your domain controllers and network. Usually, you only need specific properties for certain users. However, returning all properties for a single user is reasonable when identifying the available properties.

It's not uncommon to run a command multiple times when prototyping it. If you anticipate running a resource-intensive query when prototyping a command, consider executing it once and storing the results in a variable. Then, you can work with the variable's contents more efficiently than repeatedly executing a resource-intensive query.

For example, the following command retrieves all properties for a user account and stores the results in a variable named `$Users`. Work with the contents of the `$Users` variable instead of running the `Get-ADUser` command multiple times. Remember, the variable's contents don't update automatically when a user's information changes in Active Directory.

PowerShell

```
$Users = Get-ADUser -Identity mike -Properties *
```

You can explore the available properties by piping the `$Users` variable to `Get-Member`.

PowerShell

```
$Users | Get-Member -MemberType Properties
```

To view specific properties such as `Name`, `LastLogonDate`, and `LastBadPasswordAttempt`, pipe the `$Users` variable to `Select-Object`. This method displays the desired properties and their values based on the contents of the `$Users` variable, eliminating the need for multiple queries to Active Directory. It's a more resource-efficient approach than repeatedly executing the `Get-ADUser` command.

```
PowerShell
```

```
$Users | Select-Object -Property Name, LastLogonDate, LastBadPasswordAttempt
```

When you query Active Directory, filter the data at the source using the **Properties** parameter of `Get-ADUser` to return only the necessary properties.

```
PowerShell
```

```
Get-ADUser -Identity mike -Properties LastLogonDate, LastBadPasswordAttempt
```

Output

```
DistinguishedName      : CN=Mike F. Robbins,CN=Users,DC=mikefrobbins,DC=com
Enabled                : True
GivenName               : Mike
LastBadPasswordAttempt :
LastLogonDate          : 11/14/2023 5:10:16 AM
Name                   : Mike F. Robbins
ObjectClass             : user
ObjectGUID              : 11c7b61f-46c3-4399-9ed0-ff4e453bc2a2
SamAccountName          : mike
SID                    : S-1-5-21-611971124-518002951-3581791498-1105
Surname                : Robbins
UserPrincipalName       : mu@mikefrobbins.com
```

Summary

In this chapter, you learned how to determine what type of object a command produces, what properties and methods are available for a command, and how to work with commands that limit the properties returned by default.

Review

1. What type of object does the `Get-Process` cmdlet produce?
2. How do you determine what the available properties are for a command?
3. What should you check for if a command exists to get something but not to set the same thing?
4. How can some commands that don't return output by default be made to generate output?
5. What should you consider doing when prototyping a command that produces a large amount of output?

References

- [Get-Member](#)
- [Viewing Object Structure \(Get-Member\)](#)
- [about_Objects](#)
- [about_Properties](#)
- [about_Methods](#)
- [No PowerShell Cmdlet to Start or Stop Something? Don't Forget to Check for Methods on the Get Cmdlets ↗](#)

Next steps

In the next chapter, you'll learn about one-liners and the pipeline.

Chapter 4 - One-Liners and the pipeline

Article • 01/08/2025

When I started learning PowerShell, I initially relied on the Graphical User Interface (GUI) for tasks that seemed too complex for simple PowerShell commands. However, as I continued to learn, I improved my skills and moved from basic one-liners to creating scripts, functions, and modules. It's important to remember that feeling overwhelmed by advanced examples online is normal. No one starts as an expert in PowerShell; we all start as beginners.

For those who primarily use the GUI for administrative tasks, install the management tools on your administrative workstation to remotely manage your servers. Whether your server uses a GUI or the Server Core OS installation, this approach is beneficial. It's a practical way to familiarize yourself with remote server management in preparation for performing administrative tasks with PowerShell.

As with the previous chapters, try these concepts in your lab environment.

One-Liners

A PowerShell one-liner is one continuous pipeline. It's a common misconception that a command on one physical line is a PowerShell one-liner, but this isn't always true.

For instance, consider the following example: the command extends over multiple physical lines, yet it's a PowerShell one-liner because it forms a continuous pipeline. Line-breaking a lengthy one-liner at the pipe symbol, a natural breaking point in PowerShell, is recommended to enhance readability and clarity. This strategic use of line breaks improves readability without disrupting the flow of the pipeline.

```
PowerShell

Get-Service |  
Where-Object CanPauseAndContinue -EQ $true |  
Select-Object -Property *
```

```
Output

Name          : LanmanWorkstation  
RequiredServices : {NSI, MRxSmb20, Bowser}  
CanPauseAndContinue : True  
CanShutdown    : False  
CanStop        : True  
DisplayName    : Workstation
```

```
DependentServices : {SessionEnv, Netlogon}
MachineName      : .
ServiceName       : LanmanWorkstation
ServicesDependedOn : {NSI, MRxSmb20, Bowser}
ServiceHandle     :
Status           : Running
ServiceType       : Win32OwnProcess, Win32ShareProcess
StartType         : Automatic
Site             :
Container        :

Name            : Netlogon
RequiredServices : {LanmanWorkstation}
CanPauseAndContinue : True
CanShutdown      : False
CanStop          : True
DisplayName      : Netlogon
DependentServices : {}
MachineName      : .
ServiceName      : Netlogon
ServicesDependedOn : {LanmanWorkstation}
ServiceHandle     :
Status           : Running
ServiceType       : Win32ShareProcess
StartType         : Automatic
Site             :
Container        :

Name            : vmicheartbeat
RequiredServices : {}
CanPauseAndContinue : True
CanShutdown      : False
CanStop          : True
DisplayName      : Hyper-V Heartbeat Service
DependentServices : {}
MachineName      : .
ServiceName      : vmicheartbeat
ServicesDependedOn : {}
ServiceHandle     :
Status           : Running
ServiceType       : Win32OwnProcess, Win32ShareProcess
StartType         : Manual
Site             :
Container        :

Name            : vmickvpexchange
RequiredServices : {}
CanPauseAndContinue : True
CanShutdown      : False
CanStop          : True
DisplayName      : Hyper-V Data Exchange Service
DependentServices : {}
MachineName      : .
ServiceName      : vmickvpexchange
ServicesDependedOn : {}
```

```
ServiceHandle      :
Status            : Running
ServiceType       : Win32OwnProcess, Win32ShareProcess
StartType         : Manual
Site              :
Container        :

Name              : vmicrdv
RequiredServices  : {}
CanPauseAndContinue : True
CanShutdown       : False
CanStop           : True
DisplayName       : Hyper-V Remote Desktop Virtualization Service
DependentServices : {}
MachineName       : .
ServiceName       : vmicrdv
ServicesDependedOn : {}
ServiceHandle     :
Status            : Running
ServiceType       : Win32OwnProcess, Win32ShareProcess
StartType         : Manual
Site              :
Container        :

Name              : vmicshutdown
RequiredServices  : {}
CanPauseAndContinue : True
CanShutdown       : False
CanStop           : True
DisplayName       : Hyper-V Guest Shutdown Service
DependentServices : {}
MachineName       : .
ServiceName       : vmicshutdown
ServicesDependedOn : {}
ServiceHandle     :
Status            : Running
ServiceType       : Win32OwnProcess, Win32ShareProcess
StartType         : Manual
Site              :
Container        :

Name              : vmicvss
RequiredServices  : {}
CanPauseAndContinue : True
CanShutdown       : False
CanStop           : True
DisplayName       : Hyper-V Volume Shadow Copy Requestor
DependentServices : {}
MachineName       : .
ServiceName       : vmicvss
ServicesDependedOn : {}
ServiceHandle     :
Status            : Running
ServiceType       : Win32OwnProcess, Win32ShareProcess
StartType         : Manual
```

```
Site          : 
Container     : 

Name          : webthreatdefsvc
RequiredServices : {RpcSs, wtd}
CanPauseAndContinue : True
CanShutdown   : True
CanStop       : True
DisplayName    : Web Threat Defense Service
DependentServices : {}
MachineName   : .
ServiceName   : webthreatdefsvc
ServicesDependedOn : {RpcSs, wtd}
ServiceHandle  :
Status        : Running
ServiceType    : Win32OwnProcess, Win32ShareProcess
StartType      : Manual
Site          : 
Container     : 

Name          : webthreatdefusersvc_644de
RequiredServices : {}
CanPauseAndContinue : True
CanShutdown   : True
CanStop       : True
DisplayName    : Web Threat Defense User Service_644de
DependentServices : {}
MachineName   : .
ServiceName   : webthreatdefusersvc_644de
ServicesDependedOn : {}
ServiceHandle  :
Status        : Running
ServiceType    : 240
StartType      : Automatic
Site          : 
Container     : 

Name          : Winmgmt
RequiredServices : {RPCSS}
CanPauseAndContinue : True
CanShutdown   : True
CanStop       : True
DisplayName    : Windows Management Instrumentation
DependentServices : {}
MachineName   : .
ServiceName   : Winmgmt
ServicesDependedOn : {RPCSS}
ServiceHandle  :
Status        : Running
ServiceType    : Win32OwnProcess, Win32ShareProcess
StartType      : Automatic
Site          : 
Container     : 
```

Natural line breaks can occur at commonly used characters, including comma (,), and opening brackets ([]), braces ({ }), and parenthesis (()). Others that aren't so common include the semicolon (;), equals sign (=), and both opening single and double quotes (',"').

Using the backtick (`) or grave accent character as a line continuation is controversial. It's best to avoid it if possible. Using a backtick following a natural line break character is a common mistake. This redundancy is unnecessary and can clutter the code.

The commands in the following example execute correctly from the PowerShell console. However, attempting to run them in the console pane of the PowerShell Integrated Scripting Environment (ISE) results in an error. This difference occurs because, unlike the PowerShell console, the console pane of the ISE doesn't automatically anticipate the continuation of a command onto the next line. To prevent this issue, press `Shift + Enter` in the console pane of the ISE instead of `Enter` when you need to extend a command across multiple lines. This key combination signals to the ISE that the command is continuing on the following line, preventing the execution that leads to errors.

```
PowerShell

Get-Service -Name w32time |
    Select-Object -Property *

Output

Name          : w32time
RequiredServices : {}
CanPauseAndContinue : False
CanShutdown      : True
CanStop         : True
DisplayName     : Windows Time
DependentServices : {}
MachineName     : .
ServiceName     : w32time
ServicesDependedOn : {}
ServiceHandle    :
Status          : Running
ServiceType      : Win32OwnProcess, Win32ShareProcess
StartType        : Manual
Site            :
Container       :
```

This next example doesn't qualify as a PowerShell one-liner because it's not one continuous pipeline. Instead, it's two separate commands placed on a single line,

separated by a semicolon. This semicolon indicates the end of one command and the beginning of another.

PowerShell

```
$Service = 'w32time'; Get-Service -Name $Service
```

Output

Status	Name	DisplayName
-----	----	-----
Running	w32time	Windows Time

Many programming and scripting languages require a semicolon at the end of each line. However, in PowerShell, semicolons at the end of lines are unnecessary and not recommended. You should avoid them for cleaner and more readable code.

Filter Left

This chapter demonstrates how to filter the results of various commands.

It's a best practice in PowerShell to filter the results as early as possible in the pipeline. Achieving this involves applying filters using parameters on the initial command, usually at the beginning of the pipeline. This is commonly referred to as *filtering left*.

To illustrate this concept, consider the following example: Use the **Name** parameter of `Get-Service` to filter the results at the beginning of the pipeline, returning only the details for the Windows Time service. This method demonstrates efficient data retrieval, ensuring you only return the necessary and relevant information.

PowerShell

```
Get-Service -Name w32time
```

Output

Status	Name	DisplayName
-----	----	-----
Running	w32time	Windows Time

It's common to see online examples of a PowerShell command being piped to the `Where-Object` cmdlet to filter its results. This technique is inefficient if an earlier command in the pipeline has a parameter to perform the filtering.

PowerShell

```
Get-Service | Where-Object Name -EQ w32time
```

Output

Status	Name	DisplayName
-----	-----	-----
Running	W32Time	Windows Time

The first example demonstrates filtering directly at the source, returning results specifically for the Windows Time service. In contrast, the second example retrieves all services and then uses another command to filter the results. This might seem insignificant in small-scale scenarios, but consider a situation involving a large dataset, like Active Directory. It's inefficient to retrieve details for thousands of user accounts only to narrow them down to a small subset. Practice *filtering left* — applying filters as early as possible in the command sequence — even in seemingly trivial cases. This habit ensures efficiency in more complex scenarios where it becomes more important.

Command sequencing for effective filtering

There's a misconception that the order of commands in PowerShell is inconsequential, but this is a misunderstanding. The sequence in which you arrange commands, particularly when filtering, is important. For example, suppose you're using `Select-Object` to choose specific properties and `Where-Object` to filter. In that case, it's essential to apply the filtering first. Failing to do so means the necessary properties might not be available in the pipeline for filtering, leading to ineffective or erroneous results.

The following example fails to produce results because the `CanPauseAndContinue` property is absent when `Select-Object` is piped to `Where-Object`. This is because the `CanPauseAndContinue` property wasn't included in the selection made by `Select-Object`. Effectively, it's excluded or filtered out.

PowerShell

```
Get-Service |  
    Select-Object -Property DisplayName, Running, Status |  
    Where-Object CanPauseAndContinue
```

Reversing the order of `Select-Object` and `Where-Object` produces the desired results.

PowerShell

```
Get-Service |  
    Where-Object CanPauseAndContinue |  
    Select-Object -Property DisplayName, Status
```

Output

DisplayName	Status
Workstation	Running
Netlogon	Running
Hyper-V Heartbeat Service	Running
Hyper-V Data Exchange Service	Running
Hyper-V Remote Desktop Virtualization Service	Running
Hyper-V Guest Shutdown Service	Running
Hyper-V Volume Shadow Copy Requestor	Running
Web Threat Defense Service	Running
Web Threat Defense User Service_644de	Running
Windows Management Instrumentation	Running

The Pipeline

As seen in many examples throughout this book, you can often use the output of one command as input for another command. In Chapter 3, `Get-Member` was used to determine what type of object a command produces.

Chapter 3 also showed using the `ParameterType` parameter of `Get-Command` to determine what commands accepted that type of input. Depending on how thorough help for a command is, it might include an **INPUTS** and **OUTPUTS** section.

The **INPUTS** section indicates that you can pipe a `ServiceController` or a `String` object to the `Stop-Service` cmdlet.

PowerShell

```
help Stop-Service -Full
```

The following output is abbreviated to show the relevant portion of the help.

Output

...

INPUTS

`System.ServiceProcess.ServiceController`

You can pipe a service object to this cmdlet.

`System.String`

You can pipe a string that contains the name of a service to this cmdlet.

OUTPUTS

None

By default, this cmdlet returns no output.

`System.ServiceProcess.ServiceController`

When you use the `PassThru` parameter, this cmdlet returns a `ServiceController` object representing the service.

...

However, it doesn't specify which parameters accept this type of input. You can determine that information by checking the different parameters in the full version of the help for the `Stop-Service` cmdlet.

PowerShell

```
help Stop-Service -Full
```

Once again, only the relevant help is shown in the following results. Notice that the **DisplayName** parameter doesn't accept pipeline input. The **InputObject** parameter accepts pipeline input by value for `ServiceController` objects. The **Name** parameter accepts pipeline input by value for `String` objects and pipeline input by **property name**.

Output

...

`-DisplayName <System.String[]>`

Specifies the display names of the services to stop. Wildcard characters are permitted.

Required?	true
Position?	named
Default value	None
Accept pipeline input?	False
Accept wildcard characters?	true

`-InputObject <System.ServiceProcess.ServiceController[]>`

Specifies `ServiceController` objects that represent the services to stop. Enter a variable that contains the objects, or type a command or expression that gets the objects.

Required?	true
Position?	0
Default value	None
Accept pipeline input?	True (ByValue)
Accept wildcard characters?	false

```
-Name <System.String[]>
    Specifies the service names of the services to stop. Wildcard
    characters are permitted.

    Required?          true
    Position?         0
    Default value     None
    Accept pipeline input?   True (ByPropertyName, ByValue)
    Accept wildcard characters? true
    ...
```

When handling pipeline input, a parameter that accepts pipeline input both **by property name** and **by value** prioritizes **by value** binding first. If this method fails, it attempts to process pipeline input **by property name**. However, the term **by value** can be misleading. A more accurate description is **by type**.

For instance, if you pipe the output of a command that generates a **ServiceController** object to `Stop-Service`, this output is bound to the **InputObject** parameter. If the piped command produces a **String** object, it associates the output with the **Name** parameter. If you pipe output from a command that doesn't produce a **ServiceController** or **String** object, but does include a property named **Name**, `Stop-Service` binds the value of the **Name** property to its **Name** parameter.

Determine what type of output the `Get-Service` command produces.

PowerShell

```
Get-Service -Name w32time | Get-Member
```

`Get-Service` produces a **ServiceController** object type.

Output

```
TypeName: System.ServiceProcess.ServiceController
```

As shown in the help for `Stop-Service` cmdlet, the **InputObject** parameter accepts **ServiceController** objects through the pipeline **by value**. This implies that when you pipe the output of the `Get-Service` cmdlet to `Stop-Service`, the **ServiceController** objects produced by `Get-Service` bind to the **InputObject** parameter of `Stop-Service`.

PowerShell

```
Get-Service -Name w32time | Stop-Service
```

Now try string input. Pipe `w32time` to `Get-Member` to confirm that it's a string.

PowerShell

```
'w32time' | Get-Member
```

Output

```
TypeName: System.String
```

The PowerShell help documentation illustrates that when you pipe a string to `Stop-Service`, it binds to the `Name` parameter **by value**. Conduct a practical test to see this in action: pipe the string `w32time` to `Stop-Service`. This example demonstrates how `Stop-Service` processes the string `w32time` as the name of the service to stop. Execute the following command to observe this binding and command execution in action.

Notice that `w32time` is enclosed in single quotes. In PowerShell, it's a best practice to use single quotes for static strings, reserving double quotes for situations where the string contains variables that require expansion. Single quotes tell PowerShell to treat the content literally without parsing for variables. This approach not only ensures accuracy in how your script interprets the string but also enhances performance, as PowerShell expends less processing effort on strings within single quotes.

PowerShell

```
'w32time' | Stop-Service
```

Create a custom object to test pipeline input by property name for the `Name` parameter of `Stop-Service`.

PowerShell

```
$customObject = [pscustomobject]@{  
    Name = 'w32time'  
}
```

The contents of the `CustomObject` variable is a `PSCustomObject` object type and it contains a property named `Name`.

PowerShell

```
$customObject | Get-Member
```

Output

```
TypeName: System.Management.Automation.PSCustomObject
```

Name	MemberType	Definition
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
Name	NoteProperty	string Name=w32time

When working with variables in PowerShell, such as `$customObject` in this example, it's important to use double quotes if you need to enclose the variable in quotes. Double quotes allow for variable expansion — PowerShell evaluates the variable and uses its value. For example, if you enclose `$customObject` in double quotes and pipe it to `Get-Member`, PowerShell processes the value of `$customObject`. In contrast, using single quotes would result in piping the literal string `$customObject` to `Get-Member`, not the value of the variable. This distinction is important for scenarios where you need to evaluate the value of variables.

When piping the contents of the `$customObject` variable to the `Stop-Service` cmdlet, the binding to the **Name** parameter occurs by **property name** rather than **by value**. This is because `$customObject` is an object that contains a property named **Name**. In this scenario, PowerShell identifies the **Name** property within `$customObject` and uses its value for the **Name** parameter of `Stop-Service`.

Create another custom object using a different property name, such as **Service**.

PowerShell

```
$customObject = [pscustomobject]@{  
    Service = 'w32time'  
}
```

An error occurs while trying to stop the `w32time` service by piping `$customObject` to `Stop-Service`. The pipeline binding fails because `$customObject` doesn't produce a `ServiceController` or `String` object and doesn't contain a **Name** property.

PowerShell

```
$customObject | Stop-Service
```

Output

```
Stop-Service : Cannot find any service with service name
'@{Service=w32time}'.
At line:1 char:17
+ $customObject | Stop-Service
+
+ CategoryInfo          : ObjectNotFound: (@{Service=w32time}:String) [
Stop-Service], ServiceCommandException
+ FullyQualifiedErrorId : NoServiceFoundForGivenName,Microsoft.PowerShe
ll.Commands.StopServiceCommand
```

When the output property names of one command don't match the pipeline input requirements of another command, you can use `Select-Object` to rename the property names so they line up correctly.

In the following example, use `Select-Object` to rename the `Service` property to a property named `Name`.

At first glance, the syntax of this example might appear complex. However, it's essential to understand that more than copying and pasting code is required to learn the syntax. Instead, take the time to type out the code manually. This hands-on practice helps you remember the syntax, and it becomes more intuitive with repeated effort. Utilizing multiple monitors or split screen can also aid in the learning process. Display the example code on one screen while actively typing and experimenting with it on another. This setup makes it easier to follow along and enhances your understanding and retention of the syntax.

PowerShell

```
$customObject |
  Select-Object -Property @{Name='Name';Expression={$_['.Service}} |
  Stop-Service
```

There are instances where you might need to use a parameter that doesn't accept pipeline input. In such cases, you can still use the output of one command as the input for another. First, capture and save the display names of a few specific Windows services into a text file. This step allows you to use the saved data as input for another command.

PowerShell

```
'Background Intelligent Transfer Service', 'Windows Time' |
  Out-File -FilePath $env:TEMP\services.txt
```

You can use parentheses to pass the output of one command as input for a parameter to another command.

PowerShell

```
Stop-Service -DisplayName (Get-Content -Path $env:TEMP\services.txt)
```

This concept is like the order of operations in Algebra. Just as mathematical operations within parentheses are computed first, the command enclosed in parentheses is executed before the outer command.

PowerShellGet

PowerShellGet, a module included with PowerShell version 5.0 and higher, provides commands to discover, install, update, and publish PowerShell modules and other items in a NuGet repository. For those using PowerShell version 3.0 and above, **PowerShellGet** is also available as a separate download.

The [PowerShell Gallery](#) is an online repository hosted by Microsoft, designed as a central hub for sharing PowerShell modules, scripts, and other resources. While Microsoft hosts the PowerShell Gallery, the PowerShell community contributes most of the available modules and scripts. Given the source of these modules and scripts, exercise caution before integrating any code from the PowerShell Gallery into your environment. Review and test downloads from the PowerShell Gallery in an isolated test environment. This process ensures the code is secure and reliable, works as expected, and safeguards your environment from potential issues or vulnerabilities arising from unvetted code.

Many organizations opt to establish their own internal, private NuGet repository. This repository serves a dual purpose. First, it acts as a secure location for storing modules developed in-house, intended solely for internal use. Secondly, it provides a vetted collection of modules sourced externally, including those from public repositories. Companies typically undertake a thorough validation process before adding these external modules to the internal repository. This process is important to ensure the modules are free from malicious content and align with the security and operational standards of the company.

Use the `Find-Module` cmdlet that's part of the **PowerShellGet** module to find a module in the PowerShell Gallery that I wrote named **MrToolkit**.

PowerShell

```
Find-Module -Name MrToolkit
```

Output

```
NuGet provider is required to continue
PowerShellGet requires NuGet provider version '2.8.5.201' or newer to
interact with NuGet-based repositories. The NuGet provider must be available
in 'C:\Program Files\PackageManagement\ProviderAssemblies' or
'C:\Users\mikefrobbins\AppData\Local\PackageManagement\ProviderAssemblies'.
You can also install the NuGet provider by running 'Install-PackageProvider
-Name NuGet -MinimumVersion 2.8.5.201 -Force'. Do you want PowerShellGet to
install and import the NuGet provider now?
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"):
```

Version	Name	Repository	Description
-----	-----	-----	-----
1.3	MrToolkit	PSGallery	Misc PowerShell Tools

The first time you use one of the commands from the **PowerShellGet** module, you're prompted to install the NuGet provider.

To install the **MrToolkit** module, pipe the previous command to `Install-Module`.

PowerShell

```
Find-Module -Name MrToolkit | Install-Module -Scope CurrentUser
```

Output

```
Untrusted repository
You are installing the modules from an untrusted repository. If you trust
this repository, change its InstallationPolicy value by running the
Set-PSRepository cmdlet. Are you sure you want to install the modules from
'https://www.powershellgallery.com/api/v2'?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "N"):
```

Since the PowerShell Gallery is an untrusted repository, it prompts you to approve the installation of the module.

Finding pipeline input the easy way

The **MrToolkit** module includes a function named `Get-MrPipelineInput`. This cmdlet is designed to provide users with a convenient method for identifying the parameters of a command capable of accepting pipeline input. Specifically, it reveals three key aspects:

- Which parameters of a command can receive pipeline input
- The type of object each parameter accepts
- Whether they accept pipeline input **by value** or **by property name**

This capability dramatically simplifies the process of understanding and utilizing the pipeline capabilities of PowerShell commands.

The information previously obtained by analyzing the help documentation can be determined using this function.

```
PowerShell

Get-MrPipelineInput -Name Stop-Service | Format-List

Output

ParameterName          : InputObject
ParameterType          : System.ServiceProcess.ServiceController[]
ValueFromPipeline      : True
ValueFromPipelineByPropertyName : False

ParameterName          : Name
ParameterType          : System.String[]
ValueFromPipeline      : True
ValueFromPipelineByPropertyName : True
```

Summary

In this chapter, you learned about the intricacies of PowerShell one-liners. You also learned that the physical line count of a command is irrelevant to its classification as a PowerShell one-liner. Additionally, you learned about key concepts such as filtering left, the pipeline, and **PowerShellGet**.

Review

1. What's a PowerShell one-liner?
2. What are some characters where natural line breaks can occur in PowerShell?
3. Why should you filter left?
4. What are the two ways that a PowerShell command can accept pipeline input?
5. Why shouldn't you trust commands found in the PowerShell Gallery?

References

- [about_Pipelines](#)
- [about_Command_Syntax](#)
- [about_Parameters](#)

- PowerShellGet: The BIG EASY way to discover, install, and update PowerShell modules ↗

Next steps

In the next chapter, you'll learn about formatting, aliases, providers, and comparison operators.

Chapter 5 - Formatting, aliases, providers, comparison

Article • 01/09/2025

Prerequisites

The `SqlServer` PowerShell module is required by some examples shown in this chapter. For more information about the `SqlServer` PowerShell module and installation instructions, see [SQL Server PowerShell overview](#). It's also used in subsequent chapters. Download and install it on your Windows lab environment computer.

Format Right

In Chapter 4, you learned to filter as far to the left as possible. The rule for manually formatting a command's output is similar to that rule, except it needs to occur as far to the right as possible.

The most common format commands are `Format-Table` and `Format-List`. `Format-Wide` and `Format-Custom` can also be used, but are less common.

As mentioned in Chapter 3, a command that returns more than four properties defaults to a list unless custom formatting is used.

```
PowerShell

Get-Service -Name w32time |
    Select-Object -Property Status, DisplayName, Can*
```

```
Output

Status : Running
DisplayName : Windows Time
CanPauseAndContinue : False
CanShutdown : True
CanStop : True
```

Use the `Format-Table` cmdlet to manually override the formatting and show the output in a table instead of a list.

```
PowerShell
```

```
Get-Service -Name w32time |  
    Select-Object -Property Status, DisplayName, Can* |  
    Format-Table
```

Output

Status	DisplayName	CanPauseAndContinue	CanShutdown	CanStop
-----	-----	-----	-----	-----
Running	Windows Time	False	True	True

The default output for `Get-Service` is three properties in a table.

PowerShell

```
Get-Service -Name w32time
```

Output

Status	Name	DisplayName
-----	-----	-----
Running	w32time	Windows Time

Use the `Format-List` cmdlet to override the default formatting and return the results in a list.

PowerShell

```
Get-Service -Name w32time | Format-List
```

Notice that simply piping `Get-Service` to `Format-List` made it return additional properties. This doesn't occur with every command because of how the format for that particular command is set up behind the scenes.

Output

Name	:	w32time
DisplayName	:	Windows Time
Status	:	Running
DependentServices	:	{}
ServicesDependedOn	:	{}
CanPauseAndContinue	:	False
CanShutdown	:	True
CanStop	:	True
ServiceType	:	Win32OwnProcess, Win32ShareProcess

The number one thing to be aware of with the format cmdlets is they produce format objects that are different than normal objects in PowerShell.

PowerShell

```
Get-Service -Name w32time | Format-List | Get-Member
```

Output

TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatStartData

Name	MemberType	Definition
-----	-----	-----
Equals	Method	bool Equals(System.Object)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
autosizeInfo	Property	Microsoft.PowerShell.C...
ClassId2e4f51ef21dd47e99d3c952918aff9cd	Property	string ClassId2e4f51ef...
groupingEntry	Property	Microsoft.PowerShell.C...
pageFooterEntry	Property	Microsoft.PowerShell.C...
pageHeaderEntry	Property	Microsoft.PowerShell.C...
shapeInfo	Property	Microsoft.PowerShell.C...

TypeName: Microsoft.PowerShell.Commands.Internal.Format.GroupStartData

Name	MemberType	Definition
-----	-----	-----
Equals	Method	bool Equals(System.Object)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd	Property	string ClassId2e4f51ef...
groupingEntry	Property	Microsoft.PowerShell.C...
shapeInfo	Property	Microsoft.PowerShell.C...

TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatEntryData

Name	MemberType	Definition
-----	-----	-----
Equals	Method	bool Equals(System.Object)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd	Property	string ClassId2e4f51ef...
formatEntryInfo	Property	Microsoft.PowerShell.C...
outOfBand	Property	bool outOfBand {get;set;}
writeStream	Property	Microsoft.PowerShell.C...

```
TypeName: Microsoft.PowerShell.Commands.Internal.Format.GroupEndData
```

Name	MemberType	Definition
-----	-----	-----
Equals	Method	bool Equals(System.Object)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd	Property	string ClassId2e4f51ef...
groupingEntry	Property	Microsoft.PowerShell.C...

```
TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatEndData
```

Name	MemberType	Definition
-----	-----	-----
Equals	Method	bool Equals(System.Object)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ToString	Method	string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd	Property	string ClassId2e4f51ef...
groupingEntry	Property	Microsoft.PowerShell.C...

What this means is format commands can't be piped to most other commands. They can be piped to some of the `Out-*` commands, but that's about it. This is why you want to perform any formatting at the very end of the line (format right).

Aliases

An alias in PowerShell is a shorter name for a command. PowerShell includes a set of built-in aliases and you can also define your own aliases.

The `Get-Alias` cmdlet is used to find aliases. If you already know the alias for a command, the `Name` parameter is used to determine what command the alias is associated with.

```
PowerShell
```

```
Get-Alias -Name gcm
```

```
Output
```

CommandType	Name	Version
-----	-----	-----
Alias	gcm -> Get-Command	

Multiple aliases can be specified for the value of the `Name` parameter.

PowerShell

```
Get-Alias -Name gcm, gm
```

Output

CommandType	Name	Version
-----	-----	-----
Alias	gcm -> Get-Command	
Alias	gm -> Get-Member	

You often see the **Name** parameter omitted since it's a positional parameter.

PowerShell

```
Get-Alias gm
```

Output

CommandType	Name	Version
-----	-----	-----
Alias	gm -> Get-Member	

If you want to find aliases for a command, you need to use the **Definition** parameter.

PowerShell

```
Get-Alias -Definition Get-Command, Get-Member
```

Output

CommandType	Name	Version
-----	-----	-----
Alias	gcm -> Get-Command	
Alias	gm -> Get-Member	

The **Definition** parameter can't be used positionally, so it must be specified.

Aliases can save you a few keystrokes, and they're fine when you type commands into the console. They shouldn't be used in scripts or any code that you're saving or sharing with others. As mentioned earlier in this book, using full cmdlet and parameter names is self-documenting and easier to understand.

Use caution when creating your own aliases because they only exist in your current PowerShell session on your computer.

Providers

A provider in PowerShell is an interface that allows file system-like access to a data store. There are several built-in providers in PowerShell.

```
PowerShell
Get-PSProvider
```

As you can see in the following results, there are built-in providers for the registry, aliases, environment variables, the file system, functions, variables, certificates, and WSMAN.

Output		
Name	Capabilities	Drives
Registry	ShouldProcess, Transactions	{HKLM, HKCU}
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess, Cr...	{C, D}
Function	ShouldProcess	{Function}
Variable	ShouldProcess	{Variable}

The actual drives that these providers use to expose their data store can be determined with the `Get-PSDrive` cmdlet. The `Get-PSDrive` cmdlet not only displays drives exposed by providers but also displays Windows logical drives, including drives mapped to network shares.

Output				
Name	Used (GB)	Free (GB)	Provider	Root
Alias			Alias	
C	18.56	107.62	FileSystem	C:\
Cert			Certificate	\
D			FileSystem	D:\
Env			Environment	

Function	Function
HKCU	Registry HKEY_CURRENT_USER
HKLM	Registry HKEY_LOCAL_MACHINE
Variable	Variable
WSMan	WSMan

Third-party modules such as the **ActiveDirectory** PowerShell module and the **SqlServer** PowerShell module both add their own PowerShell provider and PSDrive.

Import the **ActiveDirectory** and **SqlServer** PowerShell modules.

PowerShell

```
Import-Module -Name ActiveDirectory, SqlServer
```

Check to see if any additional PowerShell providers were added.

PowerShell

```
Get-PSPrinter
```

Notice that in the following set of results, two new PowerShell providers now exist, one for Active Directory and another one for SQL Server.

Output

Name	Capabilities	Drives
-----	-----	-----
Registry	ShouldProcess, Transactions	{HKLM, HKCU}
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess, Credentials	{C, A, D}
Function	ShouldProcess	{Function}
Variable	ShouldProcess	{Variable}
ActiveDirectory	Include, Exclude, Filter, Shoul...	{AD}
SqlServer	Credentials	{SQLSERVER}

A PSDrive for each of those modules was also added.

PowerShell

```
Get-PSDrive
```

Output

Name	Used (GB)	Free (GB)	Provider	Root
A			FileSystem	A:\
AD			ActiveDire...	//RootDSE/
Alias			Alias	
C	19.38	107.13	FileSystem	C:\
Cert			Certificate	\
D			FileSystem	D:\
Env			Environment	
Function			Function	
HKCU			Registry	HKEY_CURRENT_USER
HKLM			Registry	HKEY_LOCAL_MACHINE
SQLSERVER			SqlServer	SQLSERVER:\
Variable			Variable	
WSMan			WSMan	

PSDrives can be accessed just like a traditional file system.

PowerShell
<code>Get-ChildItem -Path Cert:\LocalMachine\CA</code>

Output
<pre>PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\CA Thumbprint Subject ----- ----- FEE449EE0E3965A5246F000E87FDE2A065FD89D4 CN=Root Agency D559A586669B08F46A30A133F8A9ED3D038E2EA8 OU=www.verisign.com/CPS Incorp..... 109F1CAED645BB78B3EA2B94C0697C740733031C CN=Microsoft Windows Hardware C...</pre>

Comparison Operators

PowerShell contains various comparison operators that are used to compare values or find values that match certain patterns. The following table contains a list of comparison operators in PowerShell.

All the operators listed in the table are case-insensitive. To make them case-sensitive, place a `c` in front of the operator. For example, `-ceq` is the case-sensitive version of the equals (`-eq`) comparison operator.

[+] Expand table

Operator	Definition
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal to
-like	Match using the * wildcard character
-notlike	Doesn't match using the * wildcard character
-match	Matches the specified regular expression
-notmatch	Doesn't match the specified regular expression
-contains	Determines if a collection contains a specified value
-notcontains	Determines if a collection doesn't contain a specific value
-in	Determines if a specified value is in a collection
-notin	Determines if a specified value isn't in a collection
-replace	Replaces the specified value

Proper case "PowerShell" is equal to lower case "powershell" using the equals comparison operator.

PowerShell

```
'PowerShell' -eq 'powershell'
```

Output

True

It's not equal using the case-sensitive version of the equals comparison operator.

PowerShell

```
'PowerShell' -ceq 'powershell'
```

Output

False

The not equal comparison operator reverses the condition.

PowerShell

```
'PowerShell' -ne 'powershell'
```

Output

False

Greater than, greater than or equal to, less than, and less than or equal all work with string or numeric values.

PowerShell

```
5 -gt 5
```

Output

False

Using greater than or equal to instead of greater than with the previous example returns the **Boolean** true since five is equal to five.

PowerShell

```
5 -ge 5
```

Output

True

Based on the results from the previous two examples, you can probably guess how both less than and less than or equal to work.

PowerShell

```
5 -lt 10
```

Output

True

The `-like` and `-match` operators can be confusing, even for experienced PowerShell users. `-like` is used with the wildcard characters `*` and `?` to perform "like" matches.

PowerShell

```
'PowerShell' -like '*shell'
```

Output

True

The `-match` operator uses a regular expression to perform the matching.

PowerShell

```
'PowerShell' -match '^.*shell$'
```

Output

True

Use the range operator to store the numbers 1 through 10 in a variable.

PowerShell

```
$Numbers = 1..10
```

Determine if the `$Numbers` variable includes 15.

PowerShell

```
$Numbers -contains 15
```

Output

False

Determine if it includes the number 10.

PowerShell

```
$Numbers -contains 10
```

Output

```
True
```

The `-notcontains` operator reverses the logic to see if the `$Numbers` variable doesn't contain a value.

PowerShell

```
$Numbers -notcontains 15
```

The previous example returns the **Boolean** true because it's true that the `$Numbers` variable doesn't contain 15.

Output

```
True
```

It does, however, contain the number 10, so it's false when tested.

PowerShell

```
$Numbers -notcontains 10
```

Output

```
False
```

The `-in` comparison operator was first introduced in PowerShell version 3.0. It's used to determine if a value is *in* an array. The `$Numbers` variable is an array since it contains multiple values.

PowerShell

```
15 -in $Numbers
```

Output

```
False
```

In other words, `-in` performs the same test as the `contains` comparison operator except from the opposite direction.

```
PowerShell
```

```
10 -in $Numbers
```

```
Output
```

```
True
```

Fifteen isn't in the `$Numbers` array, so `false` is returned in the following example.

```
PowerShell
```

```
15 -in $Numbers
```

```
Output
```

```
False
```

Just like the `-contains` operator, `not` reverses the logic for the `-in` operator.

```
PowerShell
```

```
10 -notin $Numbers
```

The previous example returns `false` because the `$Numbers` array does include 10 and the condition tests to determine if it doesn't contain 10.

```
Output
```

```
False
```

Determine if fifteen isn't in the `$Numbers` array.

```
PowerShell
```

```
15 -notin $Numbers
```

15 is "not in" the `$Numbers` array so it returns the **Boolean** true.

Output

True

The `-replace` operator does just what you would think. It's used to replace something. Specifying one value replaces that value with nothing. In the following example, you replace "Shell" with nothing.

PowerShell

```
'PowerShell' -replace 'Shell'
```

Output

Power

If you want to replace a value with a different one, specify the new one after the pattern you want to replace. SQL Saturday in Baton Rouge is an event I try to speak at every year. In the following example, the word "Saturday" is replaced with the abbreviation "Sat".

PowerShell

```
'SQL Saturday - Baton Rouge' -replace 'saturday', 'Sat'
```

Output

SQL Sat - Baton Rouge

There are also methods like **Replace()** that can be used to replace things similar to how the replace operator works. However, the `-replace` operator is case-insensitive by default, and the **Replace()** method is case-sensitive.

PowerShell

```
'SQL Saturday - Baton Rouge'.Replace('saturday', 'Sat')
```

Notice that the word "Saturday" isn't replaced. This is because it's specified in a different case than the original.

Output

```
SQL Saturday - Baton Rouge
```

When the word "Saturday" is specified in the same case as the original, the `Replace()` method performs the replacement as expected.

PowerShell

```
'SQL Saturday - Baton Rouge'.Replace('Saturday', 'Sat')
```

Output

```
SQL Sat - Baton Rouge
```

Be careful when using methods to transform data because you can encounter unforeseen problems, such as failing the *Turkey Test*. For an example, see my blog article, [Using Pester to Test PowerShell Code with Other Cultures](#). I recommend using operators instead of methods whenever possible to avoid these types of problems.

While the comparison operators can be used, as shown in the previous examples, I typically use them with the `Where-Object` cmdlet to perform filtering.

Summary

You learned several topics in this chapter, including Formatting Right, Aliases, Providers, and Comparison Operators.

Review

1. Why is it necessary to perform formatting as far to the right as possible?
2. How do you determine what the actual cmdlet is for the `%` alias?
3. Why shouldn't you use aliases in scripts you save or code you share with others?
4. Perform a directory listing on the drives that are associated with the Registry provider.
5. What's one of the main benefits of using the replace operator instead of the replace method?

References

- [Format-Table](#)
- [Format-List](#)
- [Format-Wide](#)
- [about_Aliases](#)
- [about_Providers](#)
- [about_Comparison_Operators](#)
- [about_Arrays](#)

Next steps

In the next chapter, you'll learn about flow control, scripting, loops, and conditional logic.

Chapter 6 - Flow control

Article • 03/25/2025

Scripting

When you move from writing PowerShell one-liners to writing scripts, it sounds more complicated than it is. A script is nothing more than the same or similar commands you run interactively in the PowerShell console, except you save them as a `.ps1` file. There are some scripting constructs that you might use, such as a `foreach` loop instead of the `ForEach-Object` cmdlet. The differences can be confusing for beginners when considering that `foreach` is both a language keyword and an alias for the `ForEach-Object` cmdlet.

Looping

One of the best aspects of PowerShell is its scalability. Once you learn how to perform a task for a single item, applying the same action to hundreds of items is almost as straightforward. Loop through the items using one of the different types of loops in PowerShell.

ForEach-Object

`ForEach-Object` is a cmdlet for iterating through items in a pipeline, such as with PowerShell one-liners. `ForEach-Object` streams the objects through the pipeline.

Although the `Module` parameter of `Get-Command` accepts multiple string values, it only accepts them via pipeline input by property name. In the following scenario, if you want to pipe two string values to `Get-Command` for use with the `Module` parameter, you need to use the `ForEach-Object` cmdlet.

PowerShell

```
'ActiveDirectory', 'SQLServer' |  
    ForEach-Object {Get-Command -Module $_} |  
        Group-Object -Property ModuleName -NoElement |  
            Sort-Object -Property Count -Descending
```

Output

```
Count Name
-----
 147 ActiveDirectory
   82 SqlServer
```

In the previous example, `$_` is the current object. Beginning with PowerShell version 3.0, `$PSItem` can be used instead of `$_`. Most experienced PowerShell users prefer using `$_` since it's backward compatible and less to type.

When using the `foreach` keyword, you must store the items in memory before iterating through them, which could be difficult if you don't know how many items you're working with.

PowerShell

```
$ComputerName = 'DC01', 'WEB01'
foreach ($Computer in $ComputerName) {
    Get-ADComputer -Identity $Computer
}
```

Output

```
DistinguishedName : CN=DC01,OU=Domain Controllers,DC=mikefrobbins,DC=com
DNSHostName      : dc01.mikefrobbins.com
Enabled          : True
Name             : DC01
ObjectClass      : computer
ObjectGUID       : c38da20c-a484-469d-ba4c-bab3fb71ae8e
SamAccountName   : DC01$
SID              : S-1-5-21-2989741381-570885089-3319121794-1001
UserPrincipalName :

DistinguishedName : CN=WEB01,CN=Computers,DC=mikefrobbins,DC=com
DNSHostName      : web01.mikefrobbins.com
Enabled          : True
Name             : WEB01
ObjectClass      : computer
ObjectGUID       : 33aa530e-1e31-40d8-8c78-76a18b673c33
SamAccountName   : WEB01$
SID              : S-1-5-21-2989741381-570885089-3319121794-1107
UserPrincipalName :
```

Many times a loop such as `foreach` or `ForEach-Object` is necessary. Otherwise you receive an error message.

PowerShell

```
Get-ADComputer -Identity 'DC01', 'WEB01'
```

Output

```
Get-ADComputer : Cannot convert 'System.Object[]' to the type
'Microsoft.ActiveDirectory.Management.ADComputer' required by parameter
'Identity'. Specified method is not supported.
At line:1 char:26
+ Get-ADComputer -Identity 'DC01', 'WEB01'
+ ~~~~~~
+ CategoryInfo          : InvalidArgument: (:) [Get-ADComputer], Param
eterBindingException
+ FullyQualifiedErrorId : CannotConvertArgument,Microsoft.ActiveDirecto
ry.Management.Commands.GetADComputer
```

Other times, you can get the same results while eliminating the loop. Consult the cmdlet help to understand your options.

PowerShell

```
'DC01', 'WEB01' | Get-ADComputer
```

Output

```
DistinguishedName : CN=DC01,OU=Domain Controllers,DC=mikefrobbins,DC=com
DNSHostName      : dc01.mikefrobbins.com
Enabled          : True
Name             : DC01
ObjectClass      : computer
ObjectGUID       : c38da20c-a484-469d-ba4c-bab3fb71ae8e
SamAccountName   : DC01$
SID              : S-1-5-21-2989741381-570885089-3319121794-1001
UserPrincipalName : 

DistinguishedName : CN=WEB01,CN=Computers,DC=mikefrobbins,DC=com
DNSHostName      : web01.mikefrobbins.com
Enabled          : True
Name             : WEB01
ObjectClass      : computer
ObjectGUID       : 33aa530e-1e31-40d8-8c78-76a18b673c33
SamAccountName   : WEB01$
SID              : S-1-5-21-2989741381-570885089-3319121794-1107
UserPrincipalName :
```

As you can see in the previous examples, the **Identity** parameter for `Get-ADComputer` only accepts a single value when provided via parameter input. However, by using the

pipeline, you can send multiple values to the command because the values are processed one at a time.

For

A `for` loop iterates while a specified condition is true. I don't use the `for` loop often, but it has uses.

PowerShell

```
for ($i = 1; $i -lt 5; $i++) {  
    Write-Output "Sleeping for $i seconds"  
    Start-Sleep -Seconds $i  
}
```

Output

```
Sleeping for 1 seconds  
Sleeping for 2 seconds  
Sleeping for 3 seconds  
Sleeping for 4 seconds
```

In the previous example, the loop iterates four times by starting with the number one and continuing as long as the counter variable `$i` is less than 5. It sleeps for a total of 10 seconds.

Do

There are two different `do` loops in PowerShell: `do until` and `do while`. `do until` runs until the specified condition is false.

The following example is a numbers game that continues until the value you guess equals the same number that the `Get-Random` cmdlet generated.

PowerShell

```
$number = Get-Random -Minimum 1 -Maximum 10  
do {  
    $guess = Read-Host -Prompt "What's your guess?"  
    if ($guess -lt $number) {  
        Write-Output 'Too low!'  
    } elseif ($guess -gt $number) {  
        Write-Output 'Too high!'  
    }  
}
```

```
}
```

```
until ($guess -eq $number)
```

Output

```
What's your guess?: 1
Too low!
What's your guess?: 2
Too low!
What's your guess?: 3
```

`Do While` is the opposite. It runs as long as the specified condition is evaluated as true.

PowerShell

```
$number = Get-Random -Minimum 1 -Maximum 10
do {
    $guess = Read-Host -Prompt "What's your guess?"
    if ($guess -lt $number) {
        Write-Output 'Too low!'
    } elseif ($guess -gt $number) {
        Write-Output 'Too high!'
    }
}
while ($guess -ne $number)
```

Output

```
What's your guess?: 1
Too low!
What's your guess?: 2
Too low!
What's your guess?: 3
Too low!
What's your guess?: 4
```

The same results are achieved with a `Do While` loop by reversing the test condition to not equals.

`do` loops always run at least once because the condition is evaluated at the end of the loop.

While

Like the `do while` loop, a `while` loop runs as long as the specified condition is true. The difference, however, is that a `while` loop evaluates the condition at the top of the loop

before any code is run. So, it doesn't run if the condition is evaluated as false.

The following example calculates what day Thanksgiving Day is on in the United States. It's always on the fourth Thursday of November. The loop starts with the 22nd day of November and adds a day, while the day of the week isn't equal to Thursday. If the 22nd is a Thursday, the loop doesn't run at all.

PowerShell

```
$date = Get-Date -Date 'November 22'  
while ($date.DayOfWeek -ne 'Thursday') {  
    $date = $date.AddDays(1)  
}  
Write-Output $date
```

Output

```
Thursday, November 23, 2017 12:00:00 AM
```

break, continue, and return

The `break` keyword is designed to exit a loop and is often used with the `switch` statement. In the following example, `break` causes the loop to end after the first iteration.

PowerShell

```
for ($i = 1; $i -lt 5; $i++) {  
    Write-Output "Sleeping for $i seconds"  
    Start-Sleep -Seconds $i  
    break  
}
```

Output

```
Sleeping for 1 seconds
```

The `continue` keyword is designed to skip to the next iteration of a loop.

The following example outputs the numbers 1, 2, 4, and 5. It skips number 3 and continues with the next iteration of the loop. Like `break`, `continue` breaks out of the loop except only for the current iteration. Execution continues with the next iteration instead of breaking out of the loop altogether and stopping.

PowerShell

```
while ($i -lt 5) {  
    $i += 1  
    if ($i -eq 3) {  
        continue  
    }  
    Write-Output $i  
}
```

Output

```
1  
2  
4  
5
```

The `return` keyword is designed to exit out of the existing scope.

Notice in the following example that `return` outputs the first result and then exits out of the loop.

PowerShell

```
$number = 1..10  
foreach ($n in $number) {  
    if ($n -ge 4) {  
        return $n  
    }  
}
```

Output

```
4
```

A more thorough explanation of the `result` statement can be found in one of my blog articles: [The PowerShell return keyword ↴](#).

Summary

In this chapter, you learned about the different types of loops that exist in PowerShell.

Review

1. What's the difference between the `ForEach-Object` cmdlet and the `foreach` statement?
2. What's the primary advantage of using a `while` loop instead of a `do while` or `do until` loop?
3. How do the `break` and `continue` statements differ?

References

- [ForEach-Object](#)
- [about_Foreach](#)
- [about_For](#)
- [about_Do](#)
- [about_While](#)
- [about_Break](#)
- [about_Continue](#)
- [about_Return](#)

Chapter 7 - Working with WMI

Article • 03/24/2025

WMI and CIM

Windows PowerShell ships by default with cmdlets for working with other technologies, such as Windows Management Instrumentation (WMI). The WMI cmdlets are deprecated and aren't available in PowerShell 6+, but are covered here as you might encounter them in older scripts running on Windows PowerShell. For new development, use the CIM cmdlets instead.

Several native WMI cmdlets exist in PowerShell without you having to install any other software or modules. `Get-Command` can be used to determine what WMI cmdlets exist in Windows PowerShell. The following results are from a Windows 11 system running PowerShell version 5.1. Your results might differ depending on the PowerShell version you're running.

PowerShell

```
Get-Command -Noun WMI*
```

Output

CommandType	Name	Version
Cmdlet	Get-WmiObject	3.1.0.0
Cmdlet	Invoke-WmiMethod	3.1.0.0
Cmdlet	Register-WmiEvent	3.1.0.0
Cmdlet	Remove-WmiObject	3.1.0.0
Cmdlet	Set-WmiInstance	3.1.0.0

The Common Information Model (CIM) cmdlets were introduced in PowerShell 3.0 and are grouped within a dedicated module. To list all available CIM cmdlets, use the `Get-Command` cmdlet with the **Module** parameter, as shown in the following example.

PowerShell

```
Get-Command -Module CimCmdlets
```

Output

CommandType	Name	Version
Cmdlet	Export-BinaryMiLog	1.0.0.0
Cmdlet	Get-CimAssociatedInstance	1.0.0.0
Cmdlet	Get-CimClass	1.0.0.0
Cmdlet	Get-CimInstance	1.0.0.0
Cmdlet	Get-CimSession	1.0.0.0
Cmdlet	Import-BinaryMiLog	1.0.0.0
Cmdlet	Invoke-CimMethod	1.0.0.0
Cmdlet	New-CimInstance	1.0.0.0
Cmdlet	New-CimSession	1.0.0.0
Cmdlet	New-CimSessionOption	1.0.0.0
Cmdlet	Register-CimIndicationEvent	1.0.0.0
Cmdlet	Remove-CimInstance	1.0.0.0
Cmdlet	Remove-CimSession	1.0.0.0
Cmdlet	Set-CimInstance	1.0.0.0

The CIM cmdlets still allow you to work with WMI, so don't be confused when someone states: "*When I query WMI with the PowerShell CIM cmdlets*".

As previously mentioned, WMI is a separate technology from PowerShell, and you're just using the CIM cmdlets to access WMI. You might find an old VBScript that uses WMI Query Language (WQL) to query WMI, such as in the following example.

VB

```

strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
& "{impersonationLevel=impersonate}!\" & strComputer & "\root\CIMV2")

Set colBIOS = objWMIService.ExecQuery _
("Select * from Win32_BIOS")

For each objBIOS in colBIOS
    Wscript.Echo "Manufacturer: " & objBIOS.Manufacturer
    Wscript.Echo "Name: " & objBIOS.Name
    Wscript.Echo "Serial Number: " & objBIOS.SerialNumber
    Wscript.Echo "SMBIOS Version: " & objBIOS.SMBIOSBIOSVersion
    Wscript.Echo "Version: " & objBIOS.Version
Next

```

You can take the WQL query from the VBScript and use it with the `Get-CimInstance` cmdlet without any modifications.

PowerShell

```
Get-CimInstance -Query 'Select * from Win32_BIOS'
```

Output

```
SMBIOSBIOSVersion : 090006
Manufacturer       : American Megatrends Inc.
Name               : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber       : 3810-1995-1654-4615-2295-2755-89
Version            : VIRTUAL - 4001628
```

The previous example isn't how I typically query WMI with PowerShell. But it works and allows you to easily migrate existing Visual Basic scripts to PowerShell. When writing a one-liner to query WMI, I use the following syntax.

PowerShell

```
Get-CimInstance -ClassName Win32_BIOS
```

Output

```
SMBIOSBIOSVersion : 090006
Manufacturer       : American Megatrends Inc.
Name               : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber       : 3810-1995-1654-4615-2295-2755-89
Version            : VIRTUAL - 4001628
```

If you only want the serial number, pipe the output to `Select-Object` and specify only the `SerialNumber` property.

PowerShell

```
Get-CimInstance -ClassName Win32_BIOS | 
    Select-Object -Property SerialNumber
```

Output

```
SerialNumber
-----
3810-1995-1654-4615-2295-2755-89
```

By default, when querying WMI, several properties that are never used are retrieved behind the scenes. It doesn't matter much when querying WMI on the local computer. But once you start querying remote computers, it's not only extra processing time to return that information but also more unnecessary information to send across the network. `Get-CimInstance` has a `Property` parameter that limits the information retrieved, making the WMI query more efficient.

PowerShell

```
Get-CimInstance -ClassName Win32_BIOS -Property SerialNumber |  
    Select-Object -Property SerialNumber
```

Output

```
SerialNumber  
-----  
3810-1995-1654-4615-2295-2755-89
```

The previous results returned an object. To return a string, use the **ExpandProperty** parameter.

PowerShell

```
Get-CimInstance -ClassName Win32_BIOS -Property SerialNumber |  
    Select-Object -ExpandProperty SerialNumber
```

Output

```
3810-1995-1654-4615-2295-2755-89
```

You could also use the dotted syntax style to return a string, eliminating the need to pipe to `Select-Object`.

PowerShell

```
(Get-CimInstance -ClassName Win32_BIOS -Property SerialNumber).SerialNumber
```

Output

```
3810-1995-1654-4615-2295-2755-89
```

Query Remote Computers with the CIM cmdlets

You should still be running PowerShell as a local admin and domain user. When you try to query information from a remote computer using the `Get-CimInstance` cmdlet, you receive an access denied error message.

PowerShell

```
Get-CimInstance -ComputerName dc01 -ClassName Win32_BIOS
```

Output

```
Get-CimInstance : Access is denied.  
At line:1 char:1  
+ Get-CimInstance -ComputerName dc01 -ClassName Win32_BIOS  
+ ~~~~~  
+ CategoryInfo          : PermissionDenied: (root\cimv2:Win32_BIOS:String) [Get-CimInstance], CimException  
+ FullyQualifiedErrorId : HRESULT 0x80070005,Microsoft.Management.Infrastructure.CimCmdlets.GetCimInstanceCommand  
+ PSComputerName        : dc01
```

Many people have security concerns regarding PowerShell, but you have the same permissions in PowerShell as in the GUI. No more and no less. The problem in the previous example is that the user running PowerShell doesn't have rights to query WMI information from the DC01 server. You could relaunch PowerShell as a domain administrator since `Get-CimInstance` doesn't have a **Credential** parameter. But that isn't a good idea because anything you run from PowerShell would run as a domain admin. Depending on the situation, that scenario could be dangerous from a security standpoint.

Using the principle of least privilege, elevate to your domain admin account on a per-command basis using the **Credential** parameter if a command has one. `Get-CimInstance` doesn't have a **Credential** parameter, so the solution in this scenario is to create a **CimSession** first. Then, use the **CimSession** instead of a computer name to query WMI on the remote computer.

PowerShell

```
$CimSession = New-CimSession -ComputerName dc01 -Credential (Get-Credential)
```

Output

```
cmdlet Get-Credential at command pipeline position 1  
Supply values for the following parameters:  
Credential
```

The CIM session was stored in a variable named `$cimSession`. Notice that you also specify the `Get-Credential` cmdlet in parentheses so that it executes first, prompting for alternate credentials, before creating the new session. I show you another more efficient

way to specify alternate credentials later in this chapter, but it's important to understand this basic concept before making it more complicated.

You can now use the CIM session created in the previous example with the `Get-CimInstance` cmdlet to query the BIOS information from WMI on the remote computer.

PowerShell

```
Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS
```

Output

```
SMBIOSBIOSVersion : 090006
Manufacturer       : American Megatrends Inc.
Name               : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber       : 0986-6980-3916-0512-6608-8243-13
Version            : VIRTUAL - 4001628
PSComputerName     : dc01
```

There are several other benefits to using CIM sessions instead of just specifying a computer name. When you run multiple queries to the same computer, using a CIM session is more efficient than using the computer name for each query. Creating a CIM session only sets up the connection once. Then, multiple queries use that same session to retrieve information. Using the computer name requires the cmdlets to set up and tear down the connection with each query.

The `Get-CimInstance` cmdlet uses the WSMAN protocol by default, which means the remote computer needs PowerShell version 3.0 or higher to connect. It's actually not the PowerShell version that matters, it's the stack version. The stack version can be determined using the `Test-WSMan` cmdlet. It needs to be version 3.0, which you find with PowerShell version 3.0 and higher.

PowerShell

```
Test-WSMan -ComputerName dc01
```

Output

```
wsmid          : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentit
                  y.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

The older WMI cmdlets use the DCOM protocol, which is compatible with older versions of Windows. However, the firewall typically blocks DCOM on newer versions of Windows. The `New-CimSessionOption` cmdlet allows you to create a DCOM protocol connection for use with `New-CimSession`. This option allows the `Get-CimInstance` cmdlet to communicate with versions of Windows as old as Windows Server 2000. This ability also means that PowerShell isn't required on the remote computer when using the `Get-CimInstance` cmdlet with a CimSession configured to use the DCOM protocol.

Create the DCOM protocol option using the `New-CimSessionOption` cmdlet and store it in a variable.

PowerShell

```
$DCOM = New-CimSessionOption -Protocol Dcom
```

For efficiency, you can store your domain administrator or elevated credentials in a variable so you don't have to constantly enter them for each command.

PowerShell

```
$Cred = Get-Credential
```

Output

```
cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
```

I have a server named SQL03 that runs Windows Server 2008 (non-R2). It's the newest Windows Server operating system that doesn't have PowerShell installed by default.

Create a `CimSession` to SQL03 using the DCOM protocol.

PowerShell

```
$CimSession = New-CimSession -ComputerName sql03 -SessionOption $DCOM -
Credential $Cred
```

Notice in the previous command that you specify the variable named `$Cred` as the value for the `Credential` parameter instead of manually entering your credentials again.

The output of the query is the same regardless of the underlying protocol.

PowerShell

```
Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS
```

Output

```
SMBIOSBIOSVersion : 090006
Manufacturer       : American Megatrends Inc.
Name               : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber       : 7237-7483-8873-8926-7271-5004-86
Version            : VIRTUAL - 4001628
PSComputerName    : sql03
```

The `Get-CimSession` cmdlet is used to see what CimSessions are currently connected and what protocols they use.

PowerShell

```
Get-CimSession
```

Output

```
Id      : 1
Name    : CimSession1
InstanceId : 80742787-e38e-41b1-a7d7-fa1369cf1402
ComputerName : dc01
Protocol   : WSMAN

Id      : 2
Name    : CimSession2
InstanceId : 8fcabd81-43cf-4682-bd53-ccce1e24aecb
ComputerName : sql03
Protocol   : DCOM
```

Retrieve and store the previously created CimSessions in a variable named `$CimSession`.

PowerShell

```
$CimSession = Get-CimSession
```

Query both computers with one command, one using the WSMAN protocol and the other with DCOM.

PowerShell

```
Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS
```

Output

```
SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 0986-6980-3916-0512-6608-8243-13
Version           : VIRTUAL - 4001628
PSComputerName    : dc01

SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 7237-7483-8873-8926-7271-5004-86
Version           : VIRTUAL - 4001628
PSComputerName    : sql03
```

One of my blog articles on WMI and CIM cmdlets features a PowerShell function that automatically detects whether to use WSMan or DCOM and then sets up the appropriate CIM session for you. For more information, see [PowerShell Function to Create CimSessions to Remote Computers with Fallback to Dcom](#).

When you finish with the CIM sessions, remove them with the `Remove-CimSession` cmdlet. To remove all CIM sessions, pipe `Get-CimSession` to `Remove-CimSession`.

PowerShell

```
Get-CimSession | Remove-CimSession
```

Summary

In this chapter, you learned about using PowerShell to work with WMI on local and remote computers. You also learned how to use the CIM cmdlets to work with remote computers using the WSMan and DCOM protocols.

Review

1. What's the difference in the WMI and CIM cmdlets?
2. By default, what protocol does the `Get-CimInstance` cmdlet use?
3. What are some benefits of using a CIM session instead of specifying a computer name with `Get-CimInstance`?

4. How do you specify an alternate protocol other than the default one for use with
`Get-CimInstance`?
5. How do you close or remove CIM sessions?

References

- [about_WMI](#)
- [about_WMI_Cmdlets](#)
- [about_WQL](#)
- [CimCmdlets Module](#)
- [Video: Using CIM Cmdlets and CIM Sessions ↗](#)

Chapter 8 - PowerShell remoting

Article • 03/26/2025

PowerShell offers several ways to run commands against remote computers. In the last chapter, you explored how to remotely query WMI using the CIM cmdlets. PowerShell also includes several cmdlets that feature a built-in **ComputerName** parameter.

As shown in the following example, you can use `Get-Command` with the **ParameterName** parameter to identify cmdlets that include a **ComputerName** parameter.

```
PowerShell

Get-Command -ParameterName ComputerName

Output

CommandType Name          Version Source
----- ----
Cmdlet      Add-Computer   3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Clear-EventLog 3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Connect-PSSession 3.0.0.0 Microsoft.PowerShell.Core
Cmdlet      Enter-PSSession 3.0.0.0 Microsoft.PowerShell.Core
Cmdlet      Get-EventLog    3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Get-HotFix     3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Get-Process    3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Get-PSSession   3.0.0.0 Microsoft.PowerShell.Core
Cmdlet      Get-Service    3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Get-WmiObject  3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Invoke-Command 3.0.0.0 Microsoft.PowerShell.Core
Cmdlet      Invoke-WmiMethod 3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Limit-EventLog 3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      New-EventLog   3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      New-PSSession  3.0.0.0 Microsoft.PowerShell.Core
Cmdlet      Receive-Job    3.0.0.0 Microsoft.PowerShell.Core
Cmdlet      Receive-PSSession 3.0.0.0 Microsoft.PowerShell.Core
Cmdlet      Register-WmiEvent 3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Remove-Computer 3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Remove-EventLog 3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Remove-PSSession 3.0.0.0 Microsoft.PowerShell.Core
Cmdlet      Remove-WmiObject 3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Rename-Computer 3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Restart-Computer 3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Send-MailMessage 3.1.0.0 Microsoft.PowerShell.Utility
Cmdlet      Set-Service    3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Set-WmiInstance 3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Show-EventLog   3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Stop-Computer   3.1.0.0 Microsoft.PowerShell.Management
```

```
Cmdlet      Test-Connection    3.1.0.0 Microsoft.PowerShell.Management  
Cmdlet      Write-EventLog     3.1.0.0 Microsoft.PowerShell.Management
```

Commands such as `Get-Process` and `Get-HotFix` include a **ComputerName** parameter, but this approach isn't the long-term direction Microsoft recommends for running commands against remote systems. Even when you find a command with a **ComputerName** parameter, it often lacks a **Credential** parameter, making it difficult to specify alternate credentials. Running PowerShell from an elevated session doesn't guarantee success, as a network firewall can block the request between your system and the remote computer.

To use the PowerShell remoting commands demonstrated in this chapter, PowerShell remoting must be enabled on the remote computer. You can enable it by running the `Enable-PSRemoting` cmdlet.

PowerShell

```
Enable-PSRemoting
```

Output

```
WinRM has been updated to receive requests.  
WinRM service type changed successfully.  
WinRM service started.
```

```
WinRM has been updated for remote management.  
WinRM firewall exception enabled.
```

One-to-one remoting

If you want an interactive remote session, one-to-one remoting is what you want. This type of remoting is provided via the `Enter-PSSession` cmdlet.

Store your domain admin credentials in the `$Cred` variable. This approach allows you to enter your credentials once and reuse them on a per-command basis as long as your current PowerShell session remains active.

PowerShell

```
$Cred = Get-Credential
```

Establish a one-to-one PowerShell remoting session to the domain controller named dc01.

PowerShell

```
Enter-PSSession -ComputerName dc01 -Credential $Cred
```

Notice the PowerShell prompt is preceded by `[dc01]`. This prefix indicates you're in an interactive session with the remote computer named dc01. Any commands you run now execute on dc01, not your local machine.

Output

```
[dc01]: PS C:\Users\Administrator\Documents>
```

Remember that you can only access the PowerShell commands and modules installed on the remote computer. If you installed other modules locally, they aren't available in the remote session.

When connected via a one-to-one interactive remoting session, it's as if you're sitting directly at the remote machine.

PowerShell

```
[dc01]: Get-Process | Get-Member
```

Output

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

Name	MemberType	Definition
Handles	AliasProperty	Handles = HandleCount
Name	AliasProperty	Name = ProcessName
NPM	AliasProperty	NPM = NonpagedSystemMemorySize64
PM	AliasProperty	PM = PagedMemorySize64
SI	AliasProperty	SI = SessionId
VM	AliasProperty	VM = VirtualMemorySize64
WS	AliasProperty	WS = WorkingSet64
Disposed	Event	System.EventHandler Disposed(Sy...)
ErrorDataReceived	Event	System.Diagnostics.DataReceived...
Exited	Event	System.EventHandler Exited(Syst...
OutputDataReceived	Event	System.Diagnostics.DataReceived...
BeginErrorReadLine	Method	void BeginErrorReadLine()
BeginOutputReadLine	Method	void BeginOutputReadLine()
CancelErrorRead	Method	void CancelErrorRead()
CancelOutputRead	Method	void CancelOutputRead()
Close	Method	void Close()
CloseMainWindow	Method	bool CloseMainWindow()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef ...
Dispose	Method	void Dispose(), void IDisposable...

Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeService()
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object InitializeLifetimeService()
Kill	Method	void Kill()
Refresh	Method	void Refresh()
Start	Method	bool Start()
ToString	Method	string ToString()
WaitForExit	Method	bool WaitForExit(int milliseconds)
WaitForInputIdle	Method	bool WaitForInputIdle(int milliseconds)
__NounName	NoteProperty	string __NounName=Process
BasePriority	Property	int BasePriority {get;}
Container	Property	System.ComponentModel.IContainer Container {get;}
EnableRaisingEvents	Property	bool EnableRaisingEvents {get;set;}
ExitCode	Property	int ExitCode {get;}
ExitTime	Property	datetime ExitTime {get;}
Handle	Property	System.IntPtr Handle {get;}
HandleCount	Property	int HandleCount {get;}
HasExited	Property	bool HasExited {get;}
Id	Property	int Id {get;}
MachineName	Property	string MachineName {get;}
MainModule	Property	System.Diagnostics.ProcessModule MainModule {get;}
MainWindowHandle	Property	System.IntPtr MainWindowHandle {get;}
MainWindowTitle	Property	string MainWindowTitle {get;}
MaxWorkingSet	Property	System.IntPtr MaxWorkingSet {get;}
MinWorkingSet	Property	System.IntPtr MinWorkingSet {get;}
Modules	Property	System.Diagnostics.ProcessModule[] Modules {get;}
NonpagedSystemMemorySize	Property	int NonpagedSystemMemorySize {get;}
NonpagedSystemMemorySize64	Property	long NonpagedSystemMemorySize64 {get;}
PagedMemorySize	Property	int PagedMemorySize {get;}
PagedMemorySize64	Property	long PagedMemorySize64 {get;}
PagedSystemMemorySize	Property	int PagedSystemMemorySize {get;}
PagedSystemMemorySize64	Property	long PagedSystemMemorySize64 {get;}
PeakPagedMemorySize	Property	int PeakPagedMemorySize {get;}
PeakPagedMemorySize64	Property	long PeakPagedMemorySize64 {get;}
PeakVirtualMemorySize	Property	int PeakVirtualMemorySize {get;}
PeakVirtualMemorySize64	Property	long PeakVirtualMemorySize64 {get;}
PeakWorkingSet	Property	int PeakWorkingSet {get;}
PeakWorkingSet64	Property	long PeakWorkingSet64 {get;}
PriorityBoostEnabled	Property	bool PriorityBoostEnabled {get;set;}
PriorityClass	Property	System.Diagnostics.ProcessPriorityClass PriorityClass {get;}
PrivateMemorySize	Property	int PrivateMemorySize {get;}
PrivateMemorySize64	Property	long PrivateMemorySize64 {get;}
PrivilegedProcessorTime	Property	timespan PrivilegedProcessorTime {get;}
ProcessName	Property	string ProcessName {get;}
ProcessorAffinity	Property	System.IntPtr ProcessorAffinity {get;}
Responding	Property	bool Responding {get;}
SafeHandle	Property	Microsoft.Win32.SafeHandles.SafeHandle SafeHandle {get;}
SessionId	Property	int SessionId {get;}
Site	Property	System.ComponentModel.ISite Site {get;}
StandardError	Property	System.IO.StreamReader StandardError {get;}
StandardInput	Property	System.IO.StreamWriter StandardInput {get;}
StandardOutput	Property	System.IO.StreamWriter StandardOutput {get;}
StartInfo	Property	System.Diagnostics.ProcessStartInfo StartInfo {get;}

StartTime	Property	datetime StartTime {get;}
SynchronizingObject	Property	System.ComponentModel.ISynchron...
Threads	Property	System.Diagnostics.ProcessThrea...
TotalProcessorTime	Property	timespan TotalProcessorTime {get;}
UserProcessorTime	Property	timespan UserProcessorTime {get;}
VirtualMemorySize	Property	int VirtualMemorySize {get;}
VirtualMemorySize64	Property	long VirtualMemorySize64 {get;}
WorkingSet	Property	int WorkingSet {get;}
WorkingSet64	Property	long WorkingSet64 {get;}
PSConfiguration	PropertySet	PSConfiguration {Name, Id, Prio...
PSResources	PropertySet	PSResources {Name, Id, Handleco...
Company	ScriptProperty	System.Object Company {get=\$thi...
CPU	ScriptProperty	System.Object CPU {get=\$this.To...
Description	ScriptProperty	System.Object Description {get=...
FileVersion	ScriptProperty	System.Object FileVersion {get=...
Path	ScriptProperty	System.Object Path {get=\$this.M...
Product	ScriptProperty	System.Object Product {get=\$thi...
ProductVersion	ScriptProperty	System.Object ProductVersion {g...

When you finish working with the remote computer, run the `Exit-PSSession` cmdlet to end the remote session.

PowerShell

```
[dc01]: Exit-PSSession
```

One-to-many remoting

While you might occasionally need to perform tasks interactively on a remote computer, PowerShell remoting becomes more powerful when you simultaneously execute commands across multiple remote systems. Use the `Invoke-Command` cmdlet to run commands on one or more remote computers at the same time.

In the following example, you query three servers for the status of the Windows Time service. The `Get-Service` cmdlet is placed inside the script block of `Invoke-Command`, meaning it executes on each remote computer.

PowerShell

```
Invoke-Command -ComputerName dc01, sql02, web01 {
    Get-Service -Name W32time
} -Credential $Cred
```

The results are returned to your local session as deserialized objects.

Output

Status	Name	DisplayName	PSComputerName
Running	W32time	Windows Time	web01
Start...	W32time	Windows Time	dc01
Running	W32time	Windows Time	sql02

To confirm the returned objects are deserialized, pipe the output to `Get-Member`.

PowerShell

```
Invoke-Command -ComputerName dc01, sql02, web01 {
    Get-Service -Name W32time
} -Credential $Cred | Get-Member
```

Output

TypeName: Deserialized.System.ServiceProcess.ServiceController

Name	MemberType	Definition
GetType	Method	type GetType()
ToString	Method	string ToString(), string ToString(strin...
Name	NoteProperty	string Name=W32time
PSComputerName	NoteProperty	string PSComputerName=dc01
PSShowComputerName	NoteProperty	bool PSShowComputerName=True
RequiredServices	NoteProperty	Deserialized.System.ServiceProcess.Servi...
RunspaceId	NoteProperty	guid RunspaceId=5ed06925-8037-43ef-9072-...
CanPauseAndContinue	Property	System.Boolean {get;set;}
CanShutdown	Property	System.Boolean {get;set;}
CanStop	Property	System.Boolean {get;set;}
Container	Property	{get;set;}
DependentServices	Property	Deserialized.System.ServiceProcess.Servi...
DisplayName	Property	System.String {get;set;}
MachineName	Property	System.String {get;set;}
ServiceHandle	Property	System.String {get;set;}
ServiceName	Property	System.String {get;set;}
ServicesDependedOn	Property	Deserialized.System.ServiceProcess.Servi...
ServiceType	Property	System.String {get;set;}
Site	Property	{get;set;}
StartType	Property	System.String {get;set;}
Status	Property	System.String {get;set;}

Notice that most methods are missing from deserialized objects. The methods are missing because these objects aren't live. They're inert snapshots of the object's state when you execute the command against the remote computer. For example, you can't start or stop a service using a deserialized object since it no longer has access to the required methods.

However, this doesn't mean you can't use methods like `Stop()` with `Invoke-Command`.

The key is that you must call the method within the remote session.

To demonstrate, stop the Windows Time service on all three remote servers by invoking the `Stop()` method remotely.

PowerShell

```
Invoke-Command -ComputerName dc01, sql02, web01 {
    (Get-Service -Name W32time).Stop()
} -Credential $Cred

Invoke-Command -ComputerName dc01, sql02, web01 {
    Get-Service -Name W32time
} -Credential $Cred
```

Output

Status	Name	DisplayName	PSComputerName
Stopped	W32time	Windows Time	web01
Stopped	W32time	Windows Time	dc01
Stopped	W32time	Windows Time	sql02

As mentioned in an earlier chapter, if there's a cmdlet available to accomplish a task, it's preferable to use it rather than calling a method directly. For example, use the `Stop-Service` cmdlet instead of the `Stop()` method to stop a service.

In the previous example, the `Stop()` method is used to make a point. Some people mistakenly believe that you can't use methods with PowerShell remoting. While it's true that you can't call methods on deserialized objects returned to your local session, you can, however, invoke them within the remote session.

PowerShell sessions

In the final example from the previous section, you ran two commands using the `Invoke-Command` cmdlet. This scenario resulted in two separate sessions being established and torn down. One for each command.

Like CIM sessions, a persistent PowerShell session allows you to run multiple commands against a remote computer without the overhead of creating a new session for each command.

Create a PowerShell session to each of the three computers you're working with in this chapter, DC01, SQL02, and WEB01.

```
PowerShell
```

```
$Session = New-PSSession -ComputerName dc01, sql02, web01 -Credential $Cred
```

Now, use the `$Session` variable to start the Windows Time service by calling its method and then verify the service status.

```
PowerShell
```

```
Invoke-Command -Session $Session {((Get-Service -Name W32time).Start())}
Invoke-Command -Session $Session {Get-Service -Name W32time}
```

```
Output
```

Status	Name	DisplayName	PSComputerName
Running	W32time	Windows Time	web01
Start...	W32time	Windows Time	dc01
Running	W32time	Windows Time	sql02

Once you create the session with alternate credentials, you don't need to specify those credentials again for each command.

Be sure to remove the sessions when you finish using them.

```
PowerShell
```

```
Get-PSSession | Remove-PSSession
```

Summary

In this chapter, you learned the fundamentals of PowerShell remoting, including running commands interactively on a single remote computer and executing commands across multiple systems using one-to-many remoting. You also explored the advantages of using persistent PowerShell sessions when running multiple commands against the same remote computer.

Review

1. How do you enable PowerShell remoting?
2. What PowerShell command do you use to start an interactive session with a remote computer?
3. What's one benefit of using a PowerShell remoting session instead of specifying the computer name with each command?
4. Can you use a PowerShell session in a one-to-one interactive remoting scenario?
5. What's the difference between the objects returned by cmdlets run locally and objects returned when the same cmdlets are executed on remote computers using `Invoke-Command`?

References

- [about_Remote](#)
- [about_Remote_Output](#)
- [about_Remote_Requirements](#)
- [about_Remote_Troubleshooting](#)
- [about_Remote_Variables](#)
- [PowerShell Remoting FAQ](#)

Chapter 9 - Functions

Article • 01/23/2025

PowerShell one-liners and scripts that have to be modified often are good candidates to turn into reusable functions.

Write functions whenever possible because they're more tool-oriented. You can add the functions to a script module, put that module in a location defined in the `$env:PSModulePath`, and call the functions without needing to locate where you saved the functions. Using the **PowerShellGet** module, it's easy to share your PowerShell modules in a NuGet repository. **PowerShellGet** ships with PowerShell version 5.0 and higher. It's also available as a separate download for PowerShell version 3.0 and higher.

Don't overcomplicate things. Keep it simple and use the most straightforward way to accomplish a task. Avoid aliases and positional parameters in any code that you reuse. Format your code for readability. Don't hardcode values; use parameters and variables. Don't write unnecessary code even if it doesn't hurt anything. It adds unnecessary complexity. Attention to detail goes a long way when writing any PowerShell code.

Naming

When naming your functions in PowerShell, use a [Pascal case](#) name with an approved verb and a singular noun. To obtain a list of approved verbs in PowerShell, run `Get-Verb`. The following example sorts the results of `Get-Verb` by the **Verb** property.

```
PowerShell
Get-Verb | Sort-Object -Property Verb
```

The **Group** property gives you an idea of how the verbs are meant to be used.

```
Output
Verb      Group
-----
Add       Common
Approve   Lifecycle
Assert    Lifecycle
Backup    Data
Block     Security
Checkpoint Data
Clear     Common
Close     Common
```

Compare	Data
Complete	Lifecycle
Compress	Data
Confirm	Lifecycle
Connect	Communications
Convert	Data
ConvertFrom	Data
ConvertTo	Data
Copy	Common
Debug	Diagnostic
Deny	Lifecycle
Disable	Lifecycle
Disconnect	Communications
Dismount	Data
Edit	Data
Enable	Lifecycle
Enter	Common
Exit	Common
Expand	Data
Export	Data
Find	Common
Format	Common
Get	Common
Grant	Security
Group	Data
Hide	Common
Import	Data
Initialize	Data
Install	Lifecycle
Invoke	Lifecycle
Join	Common
Limit	Data
Lock	Common
Measure	Diagnostic
Merge	Data
Mount	Data
Move	Common
New	Common
Open	Common
Optimize	Common
Out	Data
Ping	Diagnostic
Pop	Common
Protect	Security
Publish	Data
Push	Common
Read	Communications
Receive	Communications
Redo	Common
Register	Lifecycle
Remove	Common
Rename	Common
Repair	Diagnostic
Request	Lifecycle
Reset	Common

Resize	Common
Resolve	Diagnostic
Restart	Lifecycle
Restore	Data
Resume	Lifecycle
Revoke	Security
Save	Data
Search	Common
Select	Common
Send	Communications
Set	Common
Show	Common
Skip	Common
Split	Common
Start	Lifecycle
Step	Common
Stop	Lifecycle
Submit	Lifecycle
Suspend	Lifecycle
Switch	Common
Sync	Data
Test	Diagnostic
Trace	Diagnostic
Unblock	Security
Undo	Common
Uninstall	Lifecycle
Unlock	Common
Unprotect	Security
Unpublish	Data
Unregister	Lifecycle
Update	Data
Use	Other
Wait	Lifecycle
Watch	Common
Write	Communications

It's important to use an approved verb for your PowerShell functions. Modules that contain functions with unapproved verbs generate a warning message when they're imported into a PowerShell session. That warning message makes your functions look unprofessional. Unapproved verbs also limit the discoverability of your functions.

A simple function

A function in PowerShell is declared with the `function` keyword followed by the function name and then an opening and closing curly brace (`{ }`). The code executed by the function is contained within those curly braces.

```
function Get-Version {
    $PSVersionTable.PSVersion
}
```

The function shown in the following example is a simple example that returns the version of PowerShell.

PowerShell

```
Get-Version
```

Output

Major	Minor	Build	Revision
-----	-----	-----	-----
5	1	14393	693

When you use a generic name for your functions, such as `Get-Version`, it could cause naming conflicts. Default commands added in the future or commands that others might write could conflict with them. Prefix the noun portion of your function names to help prevent naming conflicts. For example: `<ApprovedVerb>-<Prefix><SingularNoun>`.

The following example uses the prefix `PS`.

PowerShell

```
function Get-PSVersion {
    $PSVersionTable.PSVersion
}
```

Other than the name, this function is identical to the previous one.

PowerShell

```
Get-PSVersion
```

Output

Major	Minor	Build	Revision
-----	-----	-----	-----
5	1	14393	693

You can still have a name conflict even when you add a prefix to the noun. I like to prefix my function nouns with my initials. Develop a standard and stick to it.

PowerShell

```
function Get-MrPSVersion {  
    $PSVersionTable.PSVersion  
}
```

This function is no different than the previous two, except for using a more unique name to try to prevent naming conflicts with other PowerShell commands.

PowerShell

```
Get-MrPSVersion
```

Output

Major	Minor	Build	Revision
5	1	14393	693

Once loaded into memory, you can see functions on the **Function** PSDrive.

PowerShell

```
Get-ChildItem -Path Function:\Get-*Version
```

Output

CommandType	Name	Version
Function	Get-Version	
Function	Get-PSVersion	
Function	Get-MrPSVersion	

If you want to remove these functions from your current session, remove them from the **Function** PSDrive or close and reopen PowerShell.

PowerShell

```
Get-ChildItem -Path Function:\Get-*Version | Remove-Item
```

Verify that the functions were indeed removed.

```
PowerShell
```

```
Get-ChildItem -Path Function:\Get-*Version
```

If the functions were loaded as part of a module, you can unload the module to remove them.

```
PowerShell
```

```
Remove-Module -Name <ModuleName>
```

The `Remove-Module` cmdlet removes PowerShell modules from memory in your current PowerShell session. It doesn't remove them from your system or disk.

Parameters

Don't statically assign values. Use parameters and variables instead. When naming your parameters, use the same name as the default cmdlets for your parameter names whenever possible.

In the following function, notice that I used `ComputerName` and not `Computer`, `ServerName`, or `Host` for the parameter name. Using `ComputerName` standardizes the parameter name to match the parameter name and case like the default cmdlets.

```
PowerShell
```

```
function Test-MrParameter {
    param (
        $ComputerName
    )
    Write-Output $ComputerName
}
```

The following function queries all commands on your system and returns the number with specific parameter names.

```
PowerShell
```

```
function Get-MrParameterCount {
    param (
        [string[]]$ParameterName
    )
}
```

```

foreach ($Parameter in $ParameterName) {
    $Results = Get-Command -ParameterName $Parameter -ErrorAction
    SilentlyContinue

    [pscustomobject]@{
        ParameterName      = $Parameter
        NumberOfCmdlets   = $Results.Count
    }
}
}

```

As you can see in the following results, 39 commands that have a **ComputerName** parameter. There aren't any commands with parameters such as **Computer**, **ServerName**, **Host**, or **Machine**.

PowerShell

```
Get-MrParameterCount -ParameterName ComputerName, Computer, ServerName,
                      Host, Machine
```

Output

ParameterName	NumberOfCmdlets
ComputerName	39
Computer	0
ServerName	0
Host	0
Machine	0

Use the same case for your parameter names as the default cmdlets. For example, use `ComputerName`, not `computername`. This naming scheme helps people familiar with PowerShell discover your functions and look and feel like the default cmdlets.

The `param` statement allows you to define one or more parameters. A comma (,) separates the parameter definitions. For more information, see [about_Functions_Advanced_Parameters](#).

Advanced functions

Turning a function into an advanced function in PowerShell is simple. One of the differences between a function and an advanced function is that advanced functions have common parameters that are automatically added. Common parameters include parameters such as **Verbose** and **Debug**.

Start with the `Test-MrParameter` function that was used in the previous section.

PowerShell

```
function Test-MrParameter {  
  
    param (  
        $ComputerName  
    )  
  
    Write-Output $ComputerName  
  
}
```

There are a couple of different ways to see the common parameters. One is by viewing the syntax with `Get-Command`.

PowerShell

```
Get-Command -Name Test-MrParameter -Syntax
```

Notice the `Test-MrParameter` function doesn't have any common parameters.

Output

```
Test-MrParameter [[ -ComputerName] <Object>]
```

Another is to drill down into the parameters property of `Get-Command`.

PowerShell

```
(Get-Command -Name Test-MrParameter).Parameters.Keys
```

Output

```
ComputerName
```

Add the `CmdletBinding` attribute to turn the function into an advanced function.

PowerShell

```
function Test-MrCmdletBinding {  
  
    [CmdletBinding()] # Turns a regular function into an advanced function  
    param (  
        $ComputerName  
    )  
}
```

```
)  
  
    Write-Output $ComputerName  
  
}
```

When you specify `CmdletBinding`, the common parameters are added automatically. `CmdletBinding` requires a `param` block, but the `param` block can be empty.

PowerShell

```
Get-Command -Name Test-MrCmdletBinding -Syntax
```

Output

```
Test-MrCmdletBinding [[-ComputerName] <Object>] [<CommonParameters>]
```

Drilling down into the `parameters` property of `Get-Command` shows the actual parameter names, including the common ones.

PowerShell

```
(Get-Command -Name Test-MrCmdletBinding).Parameters.Keys
```

Output

```
ComputerName
Verbose
Debug
ErrorAction
WarningAction
InformationAction
ErrorVariable
WarningVariable
InformationVariable
OutVariable
OutBuffer
PipelineVariable
```

SupportsShouldProcess

The `SupportsShouldProcess` attribute adds the `WhatIf` and `Confirm` risk mitigation parameters. These parameters are only needed for commands that make changes.

PowerShell

```
function Test-MrSupportsShouldProcess {  
  
    [CmdletBinding(SupportsShouldProcess)]  
    param (  
        $ComputerName  
    )  
  
    Write-Output $ComputerName  
  
}
```

Notice that there are now **WhatIf** and **Confirm** parameters.

PowerShell

```
Get-Command -Name Test-MrSupportsShouldProcess -Syntax
```

Output

```
Test-MrSupportsShouldProcess [[-ComputerName] <Object>] [-WhatIf] [-Confirm]  
[<CommonParameters>]
```

Once again, you can also use `Get-Command` to return a list of the actual parameter names, including the common, ones along with **WhatIf** and **Confirm**.

PowerShell

```
(Get-Command -Name Test-MrSupportsShouldProcess).Parameters.Keys
```

Output

```
ComputerName  
Verbose  
Debug  
ErrorAction  
WarningAction  
InformationAction  
ErrorVariable  
WarningVariable  
InformationVariable  
OutVariable  
OutBuffer  
PipelineVariable  
WhatIf  
Confirm
```

Parameter validation

Validate input early on. Don't allow your code to continue on a path when it can't complete without valid input.

Always specify a datatype for the variables used for parameters. In the following example, **String** is specified as the datatype for the **ComputerName** parameter. This validation limits it to only allow a single computer name to be specified for the **ComputerName** parameter.

PowerShell

```
function Test-MrParameterValidation {  
  
    [CmdletBinding()]  
    param (  
        [string]$ComputerName  
    )  
  
    Write-Output $ComputerName  
  
}
```

An error is generated if more than one computer name is specified.

PowerShell

```
Test-MrParameterValidation -ComputerName Server01, Server02
```

Output

```
Test-MrParameterValidation : Cannot process argument transformation on  
parameter 'ComputerName'. Cannot convert value to type System.String.  
At line:1 char:42  
+ Test-MrParameterValidation -ComputerName Server01, Server02  
+                                         ~~~~~~  
+ CategoryInfo          : InvalidData: (:) [Test-MrParameterValidation]  
, ParameterBindingArgumentTransformationException  
+ FullyQualifiedErrorId : ParameterArgumentTransformationError,Test-MrP  
arameterValidation
```

The problem with the current definition is that it's valid to omit the value of the **ComputerName** parameter, but a value is required for the function to complete successfully. This scenario is where the **Mandatory** parameter attribute is beneficial.

The syntax used in the following example is compatible with PowerShell version 3.0 and higher. `[Parameter(Mandatory=$true)]` could be specified to make the function compatible with PowerShell version 2.0 or higher.

PowerShell

```
function Test-MrParameterValidation {  
  
    [CmdletBinding()]  
    param (  
        [Parameter(Mandatory)]  
        [string]$ComputerName  
    )  
  
    Write-Output $ComputerName  
  
}
```

Now that the **ComputerName** is required, if one isn't specified, the function prompts for one.

PowerShell

```
Test-MrParameterValidation
```

Output

```
cmdlet Test-MrParameterValidation at command pipeline position 1  
Supply values for the following parameters:  
ComputerName:
```

If you want to allow more than one value for the **ComputerName** parameter, use the **String** datatype but add square brackets (`[]`) to the datatype to allow an array of strings.

PowerShell

```
function Test-MrParameterValidation {  
  
    [CmdletBinding()]  
    param (  
        [Parameter(Mandatory)]  
        [string[]]$ComputerName  
    )  
  
    Write-Output $ComputerName
```

```
}
```

Maybe you want to specify a default value for the `ComputerName` parameter if one isn't specified. The problem is that default values can't be used with mandatory parameters. Instead, use the `ValidateNotNullOrEmpty` parameter validation attribute with a default value.

Even when setting a default value, try not to use static values. In the following example, `$env:COMPUTERNAME` is used as the default value, which is automatically translated to the local computer name if a value isn't provided.

PowerShell

```
function Test-MrParameterValidation {  
  
    [CmdletBinding()]  
    param (  
        [ValidateNotNullOrEmpty()]  
        [string[]]$ComputerName = $env:COMPUTERNAME  
    )  
  
    Write-Output $ComputerName  
  
}
```

Verbose output

Inline comments are useful if you're writing complex code, but users don't see them unless they look at the code.

The function in the following example has an inline comment in the `foreach` loop. While this particular comment might not be difficult to locate, imagine if the function contained hundreds of lines of code.

PowerShell

```
function Test-MrVerboseOutput {  
  
    [CmdletBinding()]  
    param (  
        [ValidateNotNullOrEmpty()]  
        [string[]]$ComputerName = $env:COMPUTERNAME  
    )  
  
    foreach ($Computer in $ComputerName) {  
        #Attempting to perform an action on $Computer <--- Don't use  
    }  
}
```

```
#inline comments like this, use Write-Verbose instead.  
Write-Output $Computer  
}  
}
```

A better option is to use `Write-Verbose` instead of inline comments.

PowerShell

```
function Test-MrVerboseOutput {  
  
    [CmdletBinding()]  
    param (  
        [ValidateNotNullOrEmpty()]  
        [string[]]$ComputerName = $env:COMPUTERNAME  
    )  
  
    foreach ($Computer in $ComputerName) {  
        Write-Verbose -Message "Attempting to perform an action on  
$Computer"  
        Write-Output $Computer  
    }  
}
```

The verbose output isn't displayed when the function is called without the `Verbose` parameter.

PowerShell

```
Test-MrVerboseOutput -ComputerName Server01, Server02
```

The verbose output is displayed when the function is called with the `Verbose` parameter.

PowerShell

```
Test-MrVerboseOutput -ComputerName Server01, Server02 -Verbose
```

Pipeline input

Extra code is necessary when you want your function to accept pipeline input. As mentioned earlier in this book, commands can accept pipeline input **by value** (by type) or **by property name**. You can write your functions like the native commands so they accept either one or both of these input types.

To accept pipeline input **by value**, specify the `ValueFromPipeline` parameter attribute for that particular parameter. You can only accept pipeline input **by value** from one parameter of each datatype. If you have two parameters that accept string input, only one of them can accept pipeline input **by value**. If you specified **by value** for both of the string parameters, the input wouldn't know which parameter to bind to. This scenario is another reason I call this type of pipeline input *by type* instead of **by value**.

Pipeline input is received one item at a time, similar to how items are handled in a `foreach` loop. A `process` block is required to process each item if your function accepts an array as input. If your function only accepts a single value as input, a `process` block isn't necessary but is recommended for consistency.

PowerShell

```
function Test-MrPipelineInput {  
  
    [CmdletBinding()]  
    param (  
        [Parameter(Mandatory,  
                  ValueFromPipeline)]  
        [string[]]$ComputerName  
    )  
  
    process {  
        Write-Output $ComputerName  
    }  
  
}
```

Accepting pipeline input **by property name** is similar, except you specify it with the `ValueFromPipelineByPropertyName` parameter attribute, and it can be specified for any number of parameters regardless of datatype. The key is the output of the command being piped in must have a property name that matches the name of the parameter or a parameter alias of your function.

PowerShell

```
function Test-MrPipelineInput {  
  
    [CmdletBinding()]  
    param (  
        [Parameter(Mandatory,  
                  ValueFromPipelineByPropertyName)]  
        [string[]]$ComputerName  
    )  
  
    process {  
        Write-Output $ComputerName  
    }  

```

```
}
```

```
}
```

`begin` and `end` blocks are optional. `begin` is specified before the `process` block and is used to perform any initial work before the items are received from the pipeline. Values that are piped in aren't accessible in the `begin` block. The `end` block is specified after the `process` block and is used for cleanup after all items piped in are processed.

Error handling

The function shown in the following example generates an unhandled exception when a computer can't be contacted.

```
PowerShell
```

```
function Test-MrErrorHandler {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    process {
        foreach ($Computer in $ComputerName) {
            Test-WSMan -ComputerName $Computer
        }
    }
}
```

There are a couple of different ways to handle errors in PowerShell. `Try/Catch` is the more modern way to handle errors.

```
PowerShell
```

```
function Test-MrErrorHandler {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )
}
```

```

process {
    foreach ($Computer in $ComputerName) {
        try {
            Test-WSMan -ComputerName $Computer
        }
        catch {
            Write-Warning -Message "Unable to connect to Computer:
$Computer"
        }
    }
}

```

Although the function shown in the previous example uses error handling, it generates an unhandled exception because the command doesn't generate a terminating error. Only terminating errors are caught. Specify the **ErrorAction** parameter with **Stop** as its value to turn a nonterminating error into a terminating one.

PowerShell

```

function Test-MrErrorHandler {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    process {
        foreach ($Computer in $ComputerName) {
            try {
                Test-WSMan -ComputerName $Computer -ErrorAction Stop
            }
            catch {
                Write-Warning -Message "Unable to connect to Computer:
$Computer"
            }
        }
    }
}

```

Don't modify the global `$ErrorActionPreference` variable unless absolutely necessary. If you change it in a local scope, it reverts to the previous value when you exit that scope.

If you're using something like .NET directly from within your PowerShell function, you can't specify the **ErrorAction** parameter on the command itself. You can change the `$ErrorActionPreference` variable just before you call the .NET method.

Comment-based help

Adding help to your functions is considered a best practice. Help allows people you share them with to know how to use them.

```
PowerShell

function Get-MrAutoStoppedService {

    <#
    .SYNOPSIS
        Returns a list of services that are set to start automatically, are not
        currently running, excluding the services that are set to delayed start.

    .DESCRIPTION
        Get-MrAutoStoppedService is a function that returns a list of services
        from the specified remote computer(s) that are set to start
        automatically, are not currently running, and it excludes the services
        that are set to start automatically with a delayed startup.

    .PARAMETER ComputerName
        The remote computer(s) to check the status of the services on.

    .PARAMETER Credential
        Specifies a user account that has permission to perform this action. The
        default is the current user.

    .EXAMPLE
        Get-MrAutoStoppedService -ComputerName 'Server1', 'Server2'

    .EXAMPLE
        'Server1', 'Server2' | Get-MrAutoStoppedService

    .EXAMPLE
        Get-MrAutoStoppedService -ComputerName 'Server1' -Credential (Get-
        Credential)

    .INPUTS
        String

    .OUTPUTS
        PSCustomObject

    .NOTES
        Author: Mike F. Robbins
        Website: https://mikefrobbins.com
        Twitter: @mikefrobbins
```

```
#>

[CmdletBinding()]
param (
)

#Function Body

}
```

When you add comment-based help to your functions, help can be retrieved for them like the default built-in commands.

All the syntax for writing a function in PowerShell can seem overwhelming for someone getting started. If you can't remember the syntax for something, open a second instance of the PowerShell Integrated Scripting Environment (ISE) on a separate monitor and view the "Cmdlet (advanced function) - Complete" snippet while typing in the code for your functions. Snippets can be accessed in the PowerShell ISE using the **Ctrl** + **J** key combination.

Summary

In this chapter, you learned the basics of writing functions in PowerShell, including how to:

- Create advanced functions
- Use parameter validation
- Use verbose output
- Support pipeline input
- Handle errors
- Create comment-based help

Review

1. How do you obtain a list of approved verbs in PowerShell?
2. How do you turn a PowerShell function into an advanced function?
3. When should **WhatIf** and **Confirm** parameters be added to your PowerShell functions?
4. How do you turn a nonterminating error into a terminating one?
5. Why should you add comment-based help to your functions?

References

- [about_Functions](#)
- [about_Functions_Advanced_Parameters](#)
- [about_CommonParameters](#)
- [about_Functions_CmdletBindingAttribute](#)
- [about_Functions_Advanced](#)
- [about_Try_Catch_Finally](#)
- [about_Comment_Based_Help](#)
- Video: PowerShell Toolmaking with Advanced Functions and Script Modules ↗

Chapter 10 - Script modules

Article • 03/27/2025

If you find yourself using the same PowerShell one-liners or scripts often, turning them into reusable tools is even more important. Packaging your functions in a script module gives them a more professional feel and makes them easier to support and share with others.

Dot-sourcing functions

One thing we didn't cover in the previous chapter is dot-sourcing functions. When you define a function in a script but not part of a module, the only way to load it into memory is by dot-sourcing its `.ps1` file.

For example, save the following function in a file named `Get-MrPSVersion.ps1`.

```
PowerShell

function Get-MrPSVersion {
    $PSVersionTable
}
```

When you run the script, it appears that nothing happens.

```
PowerShell

.\Get-MrPSVersion.ps1
```

Attempting to call the function results in an error because it isn't loaded into memory.

```
PowerShell

Get-MrPSVersion
```

Output

```
Get-MrPSVersion : The term 'Get-MrPSVersion' is not recognized as the name
of a cmdlet, function, script file, or operable program. Check the spelling
of the name, or if a path was included, verify that the path is correct and
try again.
At line:1 char:1
+ Get-MrPSVersion
+ ~~~~~~
```

```
+ CategoryInfo          : ObjectNotFound: (Get-MrPSVersion:String) [],
CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

You can confirm whether functions are loaded into memory by verifying their existence on the **Function:** PSDrive.

PowerShell

```
Get-ChildItem -Path Function:\Get-MrPSVersion
```

Output

```
Get-ChildItem : Cannot find path 'Get-MrPSVersion' because it does not
exist.
At line:1 char:1
+ Get-ChildItem -Path Function:\Get-MrPSVersion
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (Get-MrPSVersion:String) [Get
-ChildItem], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.Ge
tChildItemCommand
```

The issue with running the script that defines the function is that it loads it into the **Script** scope. Once the script finishes executing, PowerShell discards that scope along with the function.

To keep the function available after the script runs, it needs to be loaded into the **Global** scope. You can accomplish this by dot-sourcing the script file. You can use a relative path for this purpose.

PowerShell

```
. .\Get-MrPSVersion.ps1
```

You can also use the full path to the script when dot-sourcing it.

PowerShell

```
. C:\Demo\Get-MrPSVersion.ps1
```

If part of the path is stored in a variable, you can combine it with the rest of the path. There's no need to use string concatenation to do this.

PowerShell

```
$Path = 'C:\'  
. $Path\Get-MrPSVersion.ps1
```

Now, if you check the **Function** PSDrive, you see the `Get-MrPSVersion` function is available.

PowerShell

```
Get-ChildItem -Path Function:\Get-MrPSVersion
```

Output

CommandType	Name	Version
-----	-----	-----
Function	Get-MrPSVersion	

Script modules

In PowerShell, a script module is simply a `.psm1` file that contains one or more functions, just like a regular script, but with a different file extension.

How do you create a script module? You might assume with a command named something like `New-Module`. That assumption is a reasonable guess, but that command actually creates a dynamic module, not a script module.

This scenario is a good reminder to always read the help documentation, even when a command name looks exactly like what you need.

PowerShell

```
help New-Module
```

Output

NAME

 New-Module

SYNOPSIS

 Creates a new dynamic module that exists only in memory.

SYNTAX

```
    New-Module [-Name] <System.String> [-ScriptBlock]  
                  <System.Management.Automation.ScriptBlock> [-ArgumentList]
```

```
<System.Object[]>] [-AsCustomObject] [-Cmdlet <System.String[]>]
[-Function <System.String[]>] [-ReturnResult] [<CommonParameters>]
```

DESCRIPTION

The `New-Module` cmdlet creates a dynamic module from a script block. The members of the dynamic module, such as functions and variables, are immediately available in the session and remain available until you close the session.

Like static modules, by default, the cmdlets and functions in a dynamic module are exported and the variables and aliases are not. However, you can use the `Export-ModuleMember` cmdlet and the parameters of `New-Module` to override the defaults.

You can also use the `AsCustomObject` parameter of `New-Module` to return

the dynamic module as a custom object. The members of the module, such as functions, are implemented as script methods of the custom object instead of being imported into the session.

Dynamic modules exist only in memory, not on disk. Like all modules, the members of dynamic modules run in a private module scope that is a child of the global scope. `Get-Module` cannot get a dynamic module, but `Get-Command` can get the exported members.

To make a dynamic module available to `Get-Module`, pipe a `New-Module` command to `Import-Module`, or pipe the module object that `New-Module` returns to `Import-Module`. This action adds the dynamic module to the `Get-Module` list, but it does not save the module to disk or make it persistent.

RELATED LINKS

Online Version: https://learn.microsoft.com/powershell/module/microsoft.powershell.core/new-module?view=powershell-5.1&WT.mc_id=ps-gethelp
`Export-ModuleMember`
`Get-Module`
`Import-Module`
`Remove-Module`
`about_Modules`

REMARKS

To see the examples, type: "Get-Help New-Module -Examples".
For more information, type: "Get-Help New-Module -Detailed".
For technical information, type: "Get-Help New-Module -Full".
For online help, type: "Get-Help New-Module -Online"

The previous chapter mentioned that functions should use approved verbs. Otherwise, PowerShell generates a warning when the module is imported.

The following example uses the `New-Module` cmdlet to create a dynamic module in memory, specifically to demonstrate what happens when you don't use an approved

verb.

PowerShell

```
New-Module -Name MyModule -ScriptBlock {  
  
    function Return-MrOsVersion {  
        Get-CimInstance -ClassName Win32_OperatingSystem |  
            Select-Object -Property @{Label='OperatingSystem';Expression=  
{$_.Caption}}  
    }  
  
    Export-ModuleMember -Function Return-MrOsVersion  
  
} | Import-Module
```

Output

```
WARNING: The names of some imported commands from the module 'MyModule'  
include  
unapproved verbs that might make them less discoverable. To find the  
commands with  
unapproved verbs, run the Import-Module command again with the Verbose  
parameter. For a  
list of approved verbs, type Get-Verb.
```

Although you used the `New-Module` cmdlet in the previous example, as mentioned before, it's not the command for creating script modules in PowerShell.

To create a script module, save your functions in a `.psm1` file. For example, save the following two functions in a file named `MyScriptModule.psm1`.

PowerShell

```
function Get-MrPSVersion {  
    $PSVersionTable  
}  
  
function Get-MrComputerName {  
    $env:COMPUTERNAME  
}
```

Try to run one of the functions.

PowerShell

```
Get-MrComputerName
```

When you call the function, you receive an error saying PowerShell can't find it. Like before, checking the **Function**: PSDrive confirms that it isn't loaded into memory.

Output

```
Get-MrComputerName : The term 'Get-MrComputerName' is not recognized as the
name of a cmdlet, function, script file, or operable program. Check the
spelling of the name, or if a path was included, verify that the path is
correct and try again.
At line:1 char:1
+ Get-MrComputerName
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (Get-MrComputerName:String) [ ]
, CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

To make the function available, you can manually import the `MyScriptModule.psm1` file using the `Import-Module` cmdlet.

PowerShell

```
Import-Module C:\MyScriptModule.psm1
```

PowerShell introduced module autoloading in version 3. To take advantage of this feature, the script module must be saved in a folder with the same base name as the `.psm1` file. That folder must be located in one of the directories specified in the `$env:PSModulePath` environment variable.

PowerShell

```
$env:PSModulePath
```

The output of `$env:PSModulePath` is difficult to read.

Output

```
C:\Users\mike-ladm\Documents\WindowsPowerShell\Modules;C:\Program Files\Wind
owsPowerShell\Modules;C:\Windows\system32\WindowsPowerShell\v1.0\Modules;C:\
Program Files (x86)\Microsoft SQL Server\130\Tools\PowerShell\Modules\
```

To make the results more readable, split the paths on the semicolon path separator so each one appears on its own line.

PowerShell

```
$env:PSModulePath -split ';'
```

The first three paths in the list are the default module locations. SQL Server Management Studio added the last path when you installed it.

Output

```
C:\Users\mike-ladm\Documents\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\Windows\system32\WindowsPowerShell\v1.0\Modules
C:\Program Files (x86)\Microsoft SQL Server\130\Tools\PowerShell\Modules\
```

For module autoloading to work, you must place the `MyScriptModule.psm1` file must in a folder named `MyScriptModule`, and that folder must reside directly inside one of the paths listed in

```
$env:PSModulePath.
```

Not all those paths are equally useful. For example, the current user path on my system isn't the first one in the list. That's because I sign in to Windows with a different account than the one I use to run PowerShell. So, it doesn't point to my user's documents folder.

The second path is the `AllUsers` path, which is where I store all of my modules.

The third path points to `C:\Windows\System32`, a protected system location. Only Microsoft should be placing modules there, as it falls under the operating system's directory structure.

Once you place the `.psm1` file in an appropriate folder within one of these paths, PowerShell automatically loads the module the first time you call one of its commands.

Module manifests

Every module should include a module manifest, which is a `.psd1` file containing metadata about the module. While the `.psd1` extension is used for manifests, not all `.psd1` files are module manifests. You can also use them for other purposes, such as defining environment data in a DSC configuration.

You can create a module manifest using the `New-ModuleManifest` cmdlet. The only required parameter is `Path`, but for the module to work correctly, you must also specify the `RootModule` parameter.

It's a best practice to include values like **Author** and **Description**, especially if you plan to publish your module to a NuGet repository using **PowerShellGet**. These fields are required in that scenario.

One quick way to tell if a module lacks a manifest is to check its version.

```
PowerShell  
  
Get-Module -Name MyScriptModule
```

A version number of `0.0` is a clear sign that the module lacks a manifest.

```
Output  
  
ModuleType Version Name ExportedCommands  
---- - - - -  
Script 0.0 MyScriptModule {Get-MrComputer...}
```

You should include all recommended details when creating a module manifest to ensure your module is well-documented and ready for sharing or publishing.

```
PowerShell  
  
$moduleManifestParams = @{  
    Path =  
    "$env:ProgramFiles\WindowsPowerShell\Modules\MyScriptModule\MyScriptModule.p  
sd1"  
    RootModule = 'MyScriptModule'  
    Author = 'Mike F. Robbins'  
    Description = 'MyScriptModule'  
    CompanyName = 'mikefrobbins.com'  
}  
  
New-ModuleManifest @moduleManifestParams
```

If you omit any values when initially creating the module manifest, you can add or update it later using the `Update-ModuleManifest` cmdlet. Avoid recreating the manifest with `New-ModuleManifest` once you create it, as doing so generates a new GUID.

Defining public and private functions

Sometimes, your module might include helper functions you don't want to expose to users. These private functions are used internally by other functions in the module but aren't exposed to users. There are a few ways to handle this scenario.

If you're not following best practices and only have a `.psm1` file without a proper module structure, your only option is to control visibility using the `Export-ModuleMember` cmdlet. This option lets you explicitly define which functions should be exposed directly from within the `.psm1` script module file, keeping everything else private by default.

In the following example, only the `Get-MrPSVersion` function is exposed to users of your module, while the `Get-MrComputerName` function remains accessible internally to other functions within the module.

```
PowerShell

function Get-MrPSVersion {
    $PSVersionTable
}

function Get-MrComputerName {
    $env:COMPUTERNAME
}

Export-ModuleMember -Function Get-MrPSVersion
```

Determine what commands are available publicly in the `MyScriptModule` module.

```
PowerShell

Get-Command -Module MyScriptModule
```


Output									
<table border="1"><thead><tr><th> CommandType</th><th>Name</th><th> Version</th></tr><tr><th>-----</th><th>---</th><th>-----</th></tr></thead><tbody><tr><td> Function</td><td>Get-MrPSVersion</td><td> 1.0</td></tr></tbody></table>	CommandType	Name	Version	-----	---	-----	Function	Get-MrPSVersion	1.0
CommandType	Name	Version							
-----	---	-----							
Function	Get-MrPSVersion	1.0							

If you add a module manifest to your module, it's a best practice to explicitly list the functions you want to export in the `FunctionsToExport` section. This option gives you control over what you expose to users from the `.psd1` module manifest file.

```
PowerShell

FunctionsToExport = 'Get-MrPSVersion'
```

You don't need to use both `Export-ModuleMember` in the `.psm1` file and the `FunctionsToExport` section in the module manifest. Either approach is enough on its own.

Summary

In this chapter, you learned how to turn your functions into a script module in PowerShell. You also explored best practices for creating script modules, including the importance of adding a module manifest to define metadata and manage exported commands.

Review

1. How do you create a script module in PowerShell?
2. Why is it important to use approved verbs for your function names?
3. How do you create a module manifest in PowerShell?
4. What are the two ways to export only specific functions from a module?
5. What conditions must be met for a module to autoload when you run one of its commands?

References

- [How to Create PowerShell Script Modules and Module Manifests ↗](#)
- [about_Modules](#)
- [New-ModuleManifest](#)
- [Export-ModuleMember](#)

Optimizing your shell experience

Article • 12/01/2022

PowerShell is a command-line shell and a scripting language used for automation.

[Wikipedia](#) includes the following description of a shell:

A shell manages the user-system interaction by prompting users for input, interpreting their input, and then handling output from the underlying operating system (much like a read-eval-print loop or [REPL](#)).

Similar to other shells like `bash` or `cmd.exe`, PowerShell allows you to run any command available on your system, not just PowerShell commands.

PowerShell commands are known as *cmdlets* (pronounced command-lets). Cmdlets are PowerShell commands, not stand-alone executables. PowerShell commands can't be run in other shells without running PowerShell first.

Features of the PowerShell command-line interface

PowerShell is a modern command shell that includes the best features of other popular shells. Unlike most shells that only accept and return text, PowerShell accepts and returns .NET objects. The shell has several features that you can use to optimize your interactive user experience.

- Robust command-line [history](#)
- [Tab completion](#) and [command prediction](#)
- Supports command and parameter [aliases](#)
- [Pipeline](#) for chaining commands
- In-console [help](#) system, similar to Unix `man` pages

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Running commands in the shell

Article • 01/23/2025

PowerShell is a command-line shell and a scripting language used for automation. Similar to other shells, like `bash` on Linux or the Windows Command Shell (`cmd.exe`), PowerShell lets you run any command available on your system, not just PowerShell commands.

Types of commands

For any shell in any operating system there are three types of commands:

- **Shell language keywords** are part of the shell's scripting language.
 - Examples of `bash` keywords include: `if`, `then`, `else`, `elif`, and `fi`.
 - Examples of `cmd.exe` keywords include: `dir`, `copy`, `move`, `if`, and `echo`.
 - Examples of PowerShell keywords include: `for`, `foreach`, `try`, `catch`, and `trap`.
- **OS-native commands** are executable files installed in the operating system. The executables can be run from any command-line shell, like PowerShell. This includes script files that may require other shells to work properly. For example, if you run a Windows batch script (`.cmd` file) in PowerShell, PowerShell runs `cmd.exe` and passes in the batch file for execution.
- **Shell environment-specific commands** are commands defined in external files that can only be used within the runtime environment of the shell. These include scripts and functions, or they can be specially compiled modules that add commands to the shell runtime. In PowerShell, these commands are known as *cmdlets* (pronounced "command-lets").

Running native commands

Any native command can be run from the PowerShell command line. Usually you run the command exactly as you would in `bash` or `cmd.exe`. The following example shows running the `grep` command in `bash` on Ubuntu Linux.

Bash

```
sdwheeler@circumflex:~$ grep sdwheeler /etc/passwd
sdwheeler:x:1000:1000:,,,:/home/sdwheeler:/bin/bash
sdwheeler@circumflex:~$ pwsh
PowerShell 7.2.6
Copyright (c) Microsoft Corporation.

https://aka.ms/powershell
Type 'help' to get help.
```

After starting PowerShell on Ubuntu, you can run the same command from the PowerShell command line:

```
PowerShell

PS /home/sdwheeler> grep sdwheeler /etc/passwd
sdwheeler:x:1000:1000:,,,:/home/sdwheeler:/bin/bash
```

Passing arguments to native commands

Most shells include features for using variables, evaluating expressions, and handling strings. But each shell does these things differently. In PowerShell, all parameters start with a hyphen (-) character. In cmd.exe, most parameters use a slash (/) character. Other command-line tools may not have a special character for parameters.

Each shell has its own way of handling and evaluating strings on the command line. When running native commands in PowerShell that expect strings to be quoted in a specific way, you may need adjust how you pass those strings.

For more information, see the following articles:

- [about_Parsing](#)
- [about_Quoting_Rules](#)

PowerShell 7.2 introduced a new experimental feature `PSNativeCommandArgumentPassing` that improved native command handling. For more information, see [\\$PSNativeCommandArgumentPassing](#).

Handling output and errors

PowerShell also has several more output streams than other shells. The bash and cmd.exe shells have `stdout` and `stderr`. PowerShell has six output streams. For more information, see [about_Redirection](#) and [about_Output_Streams](#).

In general, the output sent to `stdout` by a native command is sent to the `Success` stream in PowerShell. Output sent to `stderr` by a native command is sent to the `Error` stream in PowerShell.

When a native command has a non-zero exit code, `$?` is set to `$false`. If the exit code is zero, `$?` is set to `$true`.

However, this changed in PowerShell 7.2. Error records redirected from native commands, like when using redirection operators (`2>&1`), aren't written to PowerShell's `$Error` variable and the preference variable `$ErrorActionPreference` doesn't affect the redirected output.

Many native commands write to `stderr` as an alternative stream for additional information. This behavior can cause confusion in PowerShell when looking through errors and the additional output information can be lost if `$ErrorActionPreference` is set to a state that mutes the output.

PowerShell 7.3 added a new experimental feature

`PSNativeCommandErrorActionPreference` that allows you to control whether output to `stderr` is treated as an error. For more information, see [\\$PSNativeCommandUseErrorActionPreference](#).

Running PowerShell commands

As previously noted, PowerShell commands are known as cmdlets. Cmdlets are collected into PowerShell modules that can be loaded on demand. Cmdlets can be written in any compiled .NET language or using the PowerShell scripting language itself.

PowerShell commands that run other commands

The PowerShell `call operator` (`&`) lets you run commands that are stored in variables and represented by strings or script blocks. You can use this to run any native command or PowerShell command. This is useful in a script when you need to dynamically construct the command-line parameters for a native command. For more information, see the [call operator](#).

The `Start-Process` cmdlet can be used to run native commands, but should only be used when you need to control how the command is executed. The cmdlet has parameters to support the following scenarios:

- Run a command using different credentials
- Hide the console window created by the new process

- Redirect `stdin`, `stdout`, and `stderr` streams
- Use a different working directory for the command

The following example runs the native command `sort.exe` with redirected input and output streams.

```
PowerShell

$processOptions = @{
    FilePath = "sort.exe"
    RedirectStandardInput = "TestSort.txt"
    RedirectStandardOutput = "Sorted.txt"
    RedirectStandardError = "SortError.txt"
    UseNewEnvironment = $true
}
Start-Process @processOptions
```

For more information, see [Start-Process](#).

On Windows, the `Invoke-Item` cmdlet performs the default action for the specified item. For example, it runs an executable file or opens a document file using the application associated with the document file type. The default action depends on the type of item and is resolved by the PowerShell provider that provides access to the item.

The following example opens the PowerShell source code repository in your default web browser.

```
PowerShell

Invoke-Item https://github.com/PowerShell/PowerShell
```

For more information, see [Invoke-Item](#).

Using tab-completion in the shell

Article • 02/08/2023

PowerShell provides completions on input to provide hints, enable discovery, and speed up input entry. Command names, parameter names, argument values and file paths can all be completed by pressing the `Tab` key.

The `Tab` key is the default key binding on Windows. `PSReadLine` also provides a `MenuComplete` function that's bound to `Ctrl + Space`. The `MenuComplete` function displays a list of matching completions below the command line.

These keybindings can be changed using `PSReadLine` cmdlets or the application that's hosting PowerShell. Keybindings can be different on non-Windows platforms. For more information, see [about_PSReadLine_Functions](#).

Built-in tab completion features

PowerShell has enabled tab completion for many aspects of the command line experience.

Filename completion

To fill in a filename or path from the available choices automatically, type part of the name and press the `Tab` key. PowerShell automatically expands the name to the first match that it finds. Pressing the `Tab` key again cycles through all the available choices with each key press.

Command and parameter name completion

The tab expansion of cmdlet names is slightly different. To use tab expansion on a cmdlet name, type the entire first part of the name (the verb) and the hyphen that follows it. You can fill in more of the name for a partial match. For example, if you type `get-co` and then press the `Tab` key, PowerShell automatically expands this to the `Get-Command` cmdlet (notice that it also changes the case of letters to their standard form). If you press `Tab` key again, PowerShell replaces this with the only other matching cmdlet name, `Get-Content`. Tab completion also works to resolve PowerShell aliases and native executables.

The following graphic shows examples of tab and menu completion.

A screenshot of a Windows PowerShell window. The title bar says "C:\Program Files\PowerShell\". The command line shows "PS D:\temp> |". The rest of the window is blank.

Other tab completion enhancements

Each new version of PowerShell includes improvements to tab completion that fix bugs and improve usability.

PowerShell 7.0

- Tab completion resolves variable assignments that are enums or are type constrained
- Tab completion expands abbreviated cmdlets and functions. For example, `i-psdf<tab>` returns `Import-PowerShellDataFile`

PowerShell 7.2

- Fix tab completion for unlocalized `about*` topics
- Fix splatting being treated as positional parameter in completions
- Add completions for Comment-based Help keywords
- Add completion for `#Requires` statements
- Add tab completion for `View` parameter of `Format-*` cmdlets
- Add support for class-based argument completers

PowerShell 7.3

- Fix tab completion within the script block specified for the `ValidateScriptAttribute`
- Added tab completion for loop labels after `break` and `continue`
- Improve Hashtable completion in multiple scenarios

- Parameter splatting
- **Arguments** parameter for `Invoke-CimMethod`
- **FilterHashtable** parameter for `Get-WinEvent`
- **Property** parameter for the CIM cmdlets
- Removes duplicates from member completion scenarios
- Support forward slashes in network share (UNC path) completion
- Improve member auto completion
- Prioritize `ValidateSet` completions over enums for parameters
- Add type inference support for generic methods with type parameters
- Improve type inference and completions
 - Allows methods to be shown in completion results for `ForEach-Object -MemberName`
 - Prevents completion on expressions that return **void** like (`[void]("")`)
 - Allows non-default Class constructors to show up when class completion is based on the AST

Other ways to enhance tab completion of command parameters

Built-in tab expansion is controlled by the internal function `TabExpansion` or `TabExpansion2`. It's possible to create functions or modules that replace the default behavior of these functions. You can find examples in the PowerShell Gallery by searching for the [TabExpansion ↗](#) keyword.

Using the `ValidateSet` or `ArgumentCompletions` attributes with parameters

The `ArgumentCompletions` attribute allows you to add tab completion values to a specific parameter. The `ArgumentCompletions` attribute is similar to `ValidateSet`. Both attributes takes a list of values to be presented when the user presses `Tab` after the parameter name. However, unlike `ValidateSet`, the values aren't validated.

For more information, see:

- [ArgumentCompletions](#)
- [ValidateSet](#)

Using the `ArgumentCompleter` attribute or `Register-ArgumentCompleter` with parameters

An argument completer is a script block or function that provides dynamic tab completion for parameter values.

The `ArgumentCompleter` attribute allows you to register a function that provides tab completion values for the parameter. The argument completer function must be available to the function containing the parameter with the `ArgumentCompleter` attribute. Usually, the function is defined in the same script or module.

For more information, see [ArgumentCompleter](#).

The `Register-ArgumentCompleter` cmdlet registers a script block as an argument completer function at run time for any command you specify. This allows you to define argument completers outside of the script or module or for native commands. For more information, see [Register-ArgumentCompleter](#).

Predictive IntelliSense in PSReadLine

PSReadLine 2.1.0 introduced the **Predictive IntelliSense** feature. Predictive IntelliSense provides suggestions for full commands based on items from your **PSReadLine** history.

PSReadLine 2.2.2 extends the power of Predictive IntelliSense by adding support for plug-in modules that use advanced logic to provide suggestions for full commands. The **Az.Tools.Predictor** module was the first plug-in for Predictive IntelliSense. It uses Machine Learning to predict what Azure PowerShell command you want to run and the parameters you want to use.

For more information, see [Using predictors](#).

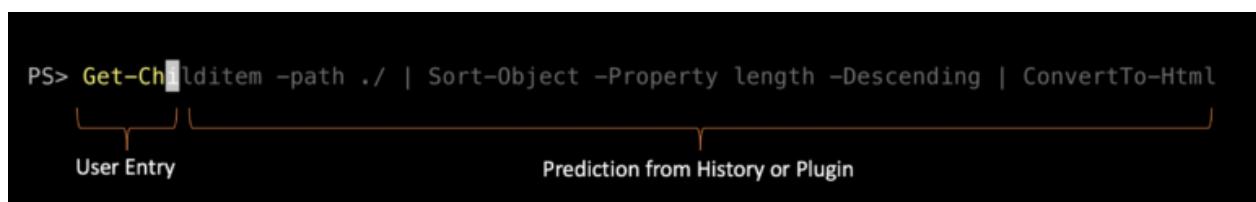
Using predictors in PSReadLine

Article • 01/11/2024

PSReadLine 2.1.0 introduced the **Predictive IntelliSense** feature. Predictive IntelliSense provides suggestions for full commands based on items from your **PSReadLine** history. PSReadLine 2.2.2 extends the power of Predictive IntelliSense by adding support for plug-in modules that use advanced logic to provide suggestions for full commands. The latest version, **PSReadLine** 2.2.6, enables predictions by default.

Using Predictive IntelliSense

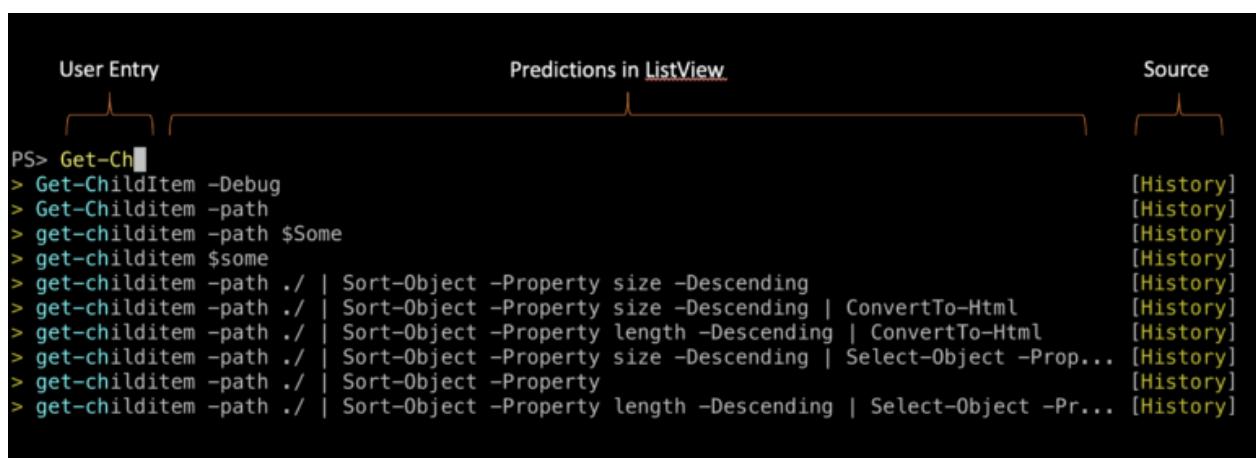
When Predictive IntelliSense is enabled, the prediction suggestion appears as colored text following the user's cursor. The suggestions from Predictive IntelliSense help new and experienced users of PowerShell discover, edit, and execute full commands based on matching predictions. Suggestions can come from the user's history and additional domain specific plugins.



A screenshot of a PowerShell window. The command entered is "PS> Get-Ch~~ilditem~~ -path ./ | Sort-Object -Property length -Descending | ConvertTo-Html". A red bracket labeled "User Entry" covers the part of the command before the cursor. A blue bracket labeled "Prediction from History or Plugin" covers the part of the command after the cursor, specifically the word "ilditem".

The previous image shows the default `InlineView` of the suggestion. Pressing `RightArrow` key accepts an inline suggestion. After accepting the suggestion, you can edit the command line before hitting `Enter` to run the command.

PSReadLine also offers a `ListView` presentation of the suggestions.



A screenshot of a PowerShell window showing the `ListView` view of Predictive IntelliSense. The command entered is "PS> Get-Ch~~ilditem~~". Below it, a list of suggestions is shown in a table:

User Entry	Predictions in <code>ListView</code>	Source
Get-Ch ilditem	Get-ChildItem -Debug	[History]
	Get-ChildItem -path	[History]
	get-childitem -path \$Some	[History]
	get-childitem \$some	[History]
	get-childitem -path ./ Sort-Object -Property size -Descending	[History]
	get-childitem -path ./ Sort-Object -Property size -Descending ConvertTo-Html	[History]
	get-childitem -path ./ Sort-Object -Property length -Descending ConvertTo-Html	[History]
	get-childitem -path ./ Sort-Object -Property size -Descending Select-Object -Prop...	[History]
	get-childitem -path ./ Sort-Object -Property	[History]
	get-childitem -path ./ Sort-Object -Property length -Descending Select-Object -Pr...	[History]

When in the list view, you can use the arrow keys to scroll through the available suggestions. List view also shows the source of the prediction.

`PSReadLine` defaults to `InlineView`. You can switch between `InlineView` and `ListView` by pressing the `F2` key. You can also use the `PredictionViewStyle` parameter of `Set-PSReadLineOption` to change the view.

Managing Predictive IntelliSense

To use Predictive IntelliSense you must have a newer version of `PSReadLine` installed. For best results, install the latest version of the module.

To install `PSReadLine` using `PowerShellGet`:

```
PowerShell  
  
Install-Module -Name PSReadLine
```

Or install using the new `PowerShellGet v3` module:

```
PowerShell  
  
Install-PSResource -Name PSReadLine
```

`PSReadLine` can be installed in Windows PowerShell 5.1 or in PowerShell 7 or higher. To use predictor plug-ins you must be running in PowerShell 7.2 or higher. Windows PowerShell 5.1 can use the history-based predictor.

In `PSReadLine` 2.2.6, Predictive IntelliSense is enabled by default depending on the following conditions:

- If Virtual Terminal (VT) is supported and `PSReadLine` running in PowerShell 7.2 or higher, `PredictionSource` is set to `HistoryAndPlugin`
- If VT is supported and `PSReadLine` running in PowerShell older than 7.2, `PredictionSource` is set to `History`
- If VT isn't supported, `PredictionSource` is set to `None`.

Use the following command to see the current setting:

```
PowerShell  
  
Get-PSReadLineOption | Select-Object -Property PredictionSource
```

You can change the prediction source using the `Set-PSReadLineOption` cmdlet with the `PredictionSource` parameter. The `PredictionSource` can be set to:

- `None`
- `History`
- `Plugin`
- `HistoryAndPlugin`

ⓘ Note

History-based predictions come from the history maintained by `PSReadLine`. That history is more comprehensive than the session-based history you can see using `Get-History`. For more information, see **Command history** section of [about_PSReadLine](#).

Setting the prediction color

By default, predictions appear in light grey text on the same line the user is typing. To support accessibility needs, you can customize the prediction color. Colors are defined using ANSI escape sequences. You can use `$PSStyle` to compose ANSI escape sequences.

PowerShell

```
Set-PSReadLineOption -Colors @{ InlinePrediction = $PSStyle.Background.Blue }
```

Or you can create your own. The default light-grey prediction text color can be restored using the following ANSI escape sequence.

PowerShell

```
Set-PSReadLineOption -Colors @{ InlinePrediction = "`e[38;5;238m" }
```

For more information about setting prediction color and other `PSReadLine` settings, see [Set-PSReadLineOption](#).

Changing keybindings

`PSReadLine` contains functions to navigate and accept predictions. For example:

- `AcceptSuggestion` - Accept the current inline suggestion
- `AcceptNextSuggestionWord` - Accept the next word of the inline suggestion

- `AcceptSuggestion` is built within `ForwardChar`, which is bound to `RightArrow` by default
- `AcceptNextSuggestionWord` is built within the function `ForwardWord`, which can be bound to `Ctrl + f`

You can use the `Set-PSReadLineKeyHandler` cmdlet to change key bindings.

PowerShell

```
Set-PSReadLineKeyHandler -Chord "Ctrl+f" -Function ForwardWord
```

With this binding, pressing `Ctrl + f` accepts the next word of an inline suggestion when the cursor is at the end of current editing line. You can bind other keys to `AcceptSuggestion` and `AcceptNextSuggestionWord` for similar functionalities. For example, you may want to make `RightArrow` accept the next word of the inline suggestion, instead of the whole suggestion line.

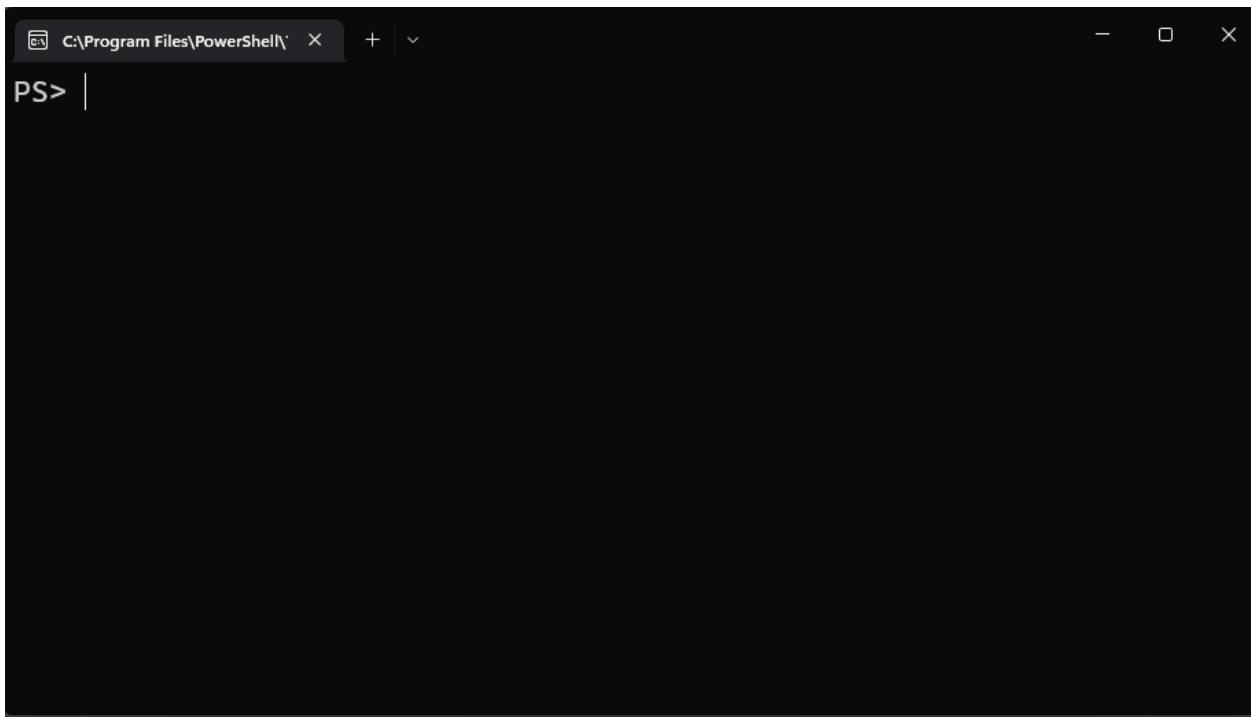
PowerShell

```
Set-PSReadLineKeyHandler -Chord "RightArrow" -Function ForwardWord
```

Using other predictor plug-ins

The `Az.Tools.Predictor` module was the first plug-in for Predictive IntelliSense. It uses Machine Learning to predict what Azure PowerShell command you want to run and the parameters you want to use. For more information and installation instructions, see [Announcing General Availability of Az.Tools.Predictor](#).

The `CompletionPredictor` module adds an IntelliSense experience for anything that can be tab-completed in PowerShell. With `PSReadLine` set to `InlineView`, you get the normal tab completion experience. When you switch to `ListView`, you get the IntelliSense experience. You can install the [CompletionPredictor](#) module from the PowerShell Gallery.



As previously noted, `ListView` shows you the source of the prediction. If you have multiple plug-ins installed the predictions are grouped by source with `History` listed first followed by each plug-in in the order that they were loaded.

Creating your own predictor module

You can write your own predictor using C# to create a compiled PowerShell module. The module must implement the

`System.Management.Automation.Subsystem.Prediction.ICommandPredictor` interface. This interface declares the methods used to query for prediction results and provide feedback.

For more information, see [How to create a command-line predictor](#).

Using dynamic help

Article • 03/27/2023

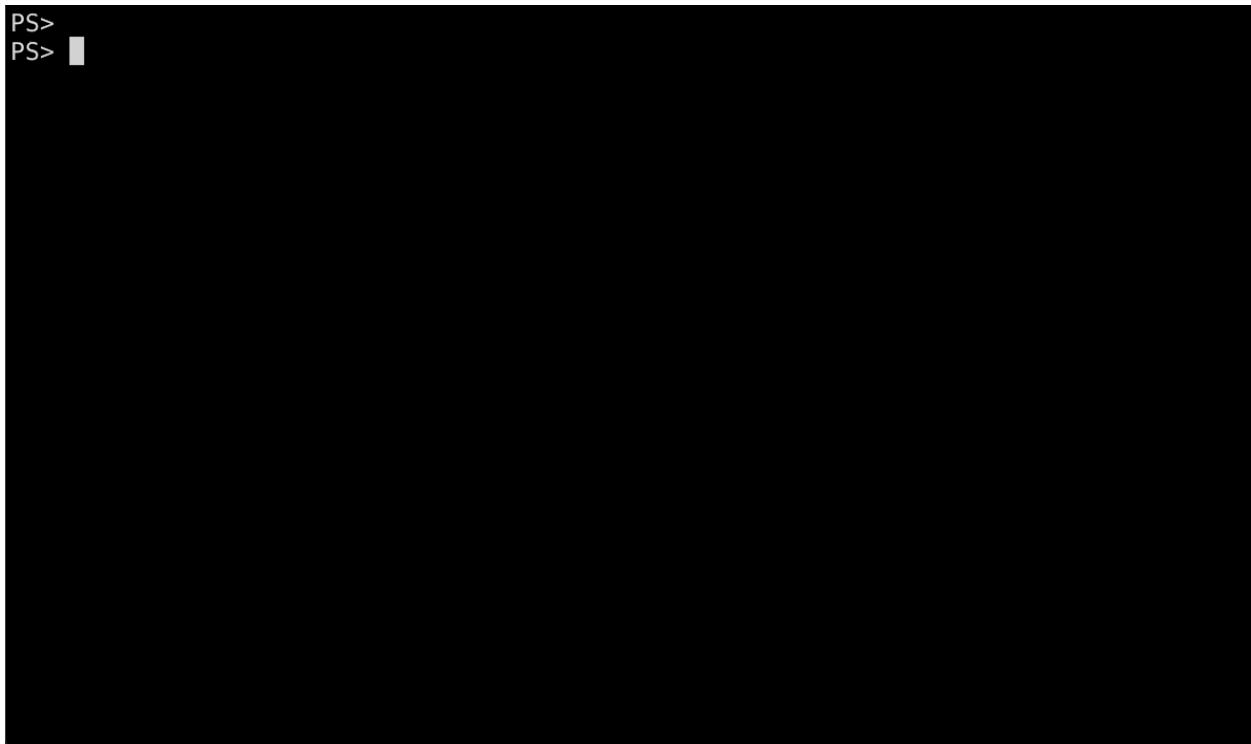
Dynamic Help provides just-in-time help that allows you to stay focused on your work without losing your place typing on the command line.

Getting cmdlet help

Dynamic Help provides a view of full cmdlet help shown in an alternative screen buffer.

PSReadLine maps the function `ShowCommandHelp` to the `F1` key.

- When the cursor is at the end of a fully expanded cmdlet name, pressing `F1` displays the help for that cmdlet.
- When the cursor is at the end of a fully expanded parameter name, pressing `F1` displays the help for the cmdlet beginning at the parameter.



The pager in **PSReadLine** allows you to scroll the displayed help using the up and down arrow keys. Pressing `Q` exits the alternative screen buffer and returns to the current cursor position on the command line on the primary screen.

Getting focused parameter help

Pressing `Alt + H` provides dynamic help for parameters. The help is shown below the current command line similar to [MenuComplete](#). The cursor must be at the end of the

fully expanded parameter name when you press the `Alt + h` key.

```
PS>
PS> Get-ChildItem -path█
-Path <System.String[]>

DESC: Specifies a path to one or more locations. Wildcards are accepted. The default
location is the current directory ('.').
Required: false, Position: 0, Default Value: Current directory, Pipeline Input: True
(ByPropertyName, ByValue), WildCard: true
```

Selecting arguments on the command line

To quickly select and edit the arguments of a cmdlet without disturbing your syntax using `Alt + a`. Based on the cursor position, it searches from the current cursor position and stops when it finds any arguments on the command line.

```
PS>
PS> get-childitem -path ./ | Sort-Object -Property length -Descending | format-table
-AutoSize█
```

Choosing keybindings

Not all keybindings work for all operating systems and terminal applications. For example, keybindings for the `Alt` key don't work on macOS by default. On Linux, `Ctrl + [` is the same as `Escape`. And `Ctrl + Spacebar` generates a `Control + 2` key sequence instead of the `Control + Spacebar` sequence expected.

To work around these quirks, map the PSReadLine function to an available key combination. For example:

```
Set-PSReadLineKeyHandler -Chord 'Ctrl+l' -Function ShowParameterHelp  
Set-PSReadLineKeyHandler -Chord 'Ctrl+k' -Function SelectCommandArgument
```

For more information about keybindings and workarounds, see [Using PSReadLine key handlers](#).

Using aliases

Article • 07/23/2024

An alias is an alternate name or shorthand name for a cmdlet or for a command element, such as a function, script, file, or executable file. You can run the command using the alias instead of the executable name.

Managing command aliases

PowerShell provides cmdlets for managing command aliases. The following command shows the cmdlets that manage aliases.

PowerShell

```
Get-Command -Noun Alias
```

Output

CommandType	Name	Version	Source
Cmdlet	Export-Alias	7.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	Get-Alias	7.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	Import-Alias	7.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	New-Alias	7.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	Remove-Alias	7.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	Set-Alias	7.0.0.0	Microsoft.PowerShell.Utility

For more information, see [about_Aliases](#).

Use the [Get-Alias](#) cmdlet to list the aliases available in your environment. To list the aliases for a single cmdlet, use the **Definition** parameter and specify the executable name.

PowerShell

```
Get-Alias -Definition Get-ChildItem
```

Output

CommandType	Name
Alias	dir -> Get-ChildItem
Alias	gci -> Get-ChildItem
Alias	ls -> Get-ChildItem

To get the definition of a single alias, use the `Name` parameter.

PowerShell

```
Get-Alias -Name gci
```

Output

CommandType	Name
Alias	gci -> Get-ChildItem

To create an alias, use the `Set-Alias` command. You can create aliases for cmdlets, functions, scripts, and native executables files.

PowerShell

```
Set-Alias -Name np -Value Notepad.exe
Set-Alias -Name cmpo -Value Compare-Object
```

Compatibility aliases in Windows

PowerShell has several aliases that allow **Unix** and `cmd.exe` users to use familiar commands in Windows. The following table show common commands, the related PowerShell cmdlet, and the PowerShell alias:

[] Expand table

Windows Command Shell	Unix command	PowerShell cmdlet	PowerShell alias
<code>cd</code> , <code>chdir</code>	<code>cd</code>	<code>Set-Location</code>	<code>sl</code> , <code>cd</code> , <code>chdir</code>
<code>cls</code>	<code>clear</code>	<code>Clear-Host</code>	<code>cls</code> <code>clear</code>
<code>copy</code>	<code>cp</code>	<code>Copy-Item</code>	<code>cpi</code> , <code>cp</code> , <code>copy</code>
<code>del</code> , <code>erase</code> , <code>rd</code> , <code>rmdir</code>	<code>rm</code>	<code>Remove-Item</code>	<code>ri</code> , <code>del</code> , <code>erase</code> , <code>rd</code> , <code>rm</code> , <code>rmdir</code>
<code>dir</code>	<code>ls</code>	<code>Get-ChildItem</code>	<code>gci</code> , <code>dir</code> , <code>ls</code>
<code>echo</code>	<code>echo</code>	<code>Write-Output</code>	<code>write</code> <code>echo</code>
<code>md</code>	<code>mkdir</code>	<code>New-Item</code>	<code>ni</code>
<code>move</code>	<code>mv</code>	<code>Move-Item</code>	<code>mi</code> , <code>move</code> , <code>mi</code>

Windows Command Shell	Unix command	PowerShell cmdlet	PowerShell alias
popd	popd	Pop-Location	popd
	pwd	Get-Location	gl, pwd
pushd	pushd	Push-Location	pushd
ren	mv	Rename-Item	rni, ren
type	cat	Get-Content	gc, cat, type

! Note

The aliases in this table are Windows-specific. Some aliases aren't available on other platforms. This is to allow the native command to work in a PowerShell session. For example, `ls` isn't defined as a PowerShell alias on macOS or Linux so that the native command is run instead of `Get-ChildItem`.

Creating alternate names for commands with parameters

You can assign an alias to a cmdlet, script, function, or executable file. Unlike some Unix shells, you cannot assign an alias to a command with parameters. For example, you can assign an alias to the `Get-EventLog` cmdlet, but you cannot assign an alias to the `Get-EventLog -LogName System` command. You must create a function that contains the command with parameters.

For more information, see [about_Aliases](#).

Parameter aliases and shorthand names

PowerShell also provides ways to create shorthand names for parameters. Parameter aliases are defined using the `Alias` attribute when you declare the parameter. These can't be defined using the `*-Alias` cmdlets.

For more information, see the [Alias attribute](#) documentation.

In addition to parameter aliases, PowerShell lets you specify the parameter name using the fewest characters needed to uniquely identify the parameter. For example, the `Get-ChildItem` cmdlet has the `Recurse` and `ReadOnly` parameters. To uniquely identify the `Recurse` parameter

you only need to provide `-Rec`. If you combine that with the command alias, `Get-ChildItem -Recurse` can be shortened to `dir -Rec`.

Don't use aliases in scripts

Aliases are a convenience feature to be used interactively in the shell. You should always use the full command and parameter names in your scripts.

- Aliases can be deleted or redefined in a profile script
- Any aliases you define may not be available to the user of your scripts
- Aliases make your code harder to read and maintain

Customizing your shell environment

Article • 02/26/2025

A PowerShell profile is a script that runs when PowerShell starts. You can use the profile to customize the environment. You can:

- Add aliases, functions, and variables
- Load modules
- Create PowerShell drives
- Run arbitrary commands
- Change preference settings

Putting these settings in your profile ensures that they're available whenever you start PowerShell on your system.

ⓘ Note

To run scripts in Windows, the PowerShell execution policy needs to be set to `RemoteSigned` at a minimum. Execution policies don't apply to macOS and Linux. For more information, see [about_Execution_Policy](#).

The `$PROFILE` variable

The `$PROFILE` automatic variable stores the paths to the PowerShell profiles that are available in the current session.

There are four possible profiles available to support different user scopes and different PowerShell hosts. The fully qualified paths for each profile script are stored in the following member properties of `$PROFILE`.

- `AllUsersAllHosts`
- `AllUsersCurrentHost`
- `CurrentUserAllHosts`
- `CurrentUserCurrentHost`

You can create profile scripts that run for all users or just one user, the `CurrentUser`.

`CurrentUser` profiles are stored under the user's home directory path. The location varies depending on the operating system and the version of PowerShell you use.

By default, referencing the `$PROFILE` variable returns the path to the "Current User, Current Host" profile. The other profiles path can be accessed through the properties of

the `$PROFILE` variable. The following command shows the default profile locations on Windows.

PowerShell

```
PS> $PROFILE | Select-Object *
AllUsersAllHosts      : C:\Program Files\PowerShell\7\profile.ps1
AllUsersCurrentHost   : C:\Program
Files\PowerShell\7\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts    : C:\Users\username\Documents\PowerShell\profile.ps1
CurrentUserCurrentHost :
C:\Users\username\Documents\PowerShell\Microsoft.PowerShell_profile.ps1
Length                : 69
```

The following command shows the default profile locations on Ubuntu Linux.

PowerShell

```
$PROFILE | Select-Object *
AllUsersAllHosts      : /opt/microsoft/powershell/7/profile.ps1
AllUsersCurrentHost   :
/opt/microsoft/powershell/7/Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts    : /home/username/.config/powershell/profile.ps1
CurrentUserCurrentHost :
/home/username/.config/powershell/Microsoft.PowerShell_profile.ps1
Length                : 67
```

There are also profiles that run for all PowerShell hosts or specific hosts. The profile script for each PowerShell host has a name unique for that host. For example, the filename for the standard Console Host on Windows or the default terminal application on other platforms is `Microsoft.PowerShell_profile.ps1`. For Visual Studio Code (VS Code), the filename is `Microsoft.VSCode_profile.ps1`.

For more information, see [about_Profiles](#).

How to create your personal profile

When you first install PowerShell on a system, the profile script files and the directories they belong to don't exist. The following command creates the "Current User, Current Host" profile script file if it doesn't exist.

PowerShell

```
if (!(Test-Path -Path $PROFILE)) {
  New-Item -ItemType File -Path $PROFILE -Force
```

```
}
```

The **Force** parameter of `New-Item` cmdlet creates the necessary folders when they don't exist. After you create the script file, you can use your favorite editor to customize your shell environment.

Adding customizations to your profile

The previous articles talked about using [tab completion](#), [command predictors](#), and [aliases](#). These articles showed the commands used to load the required modules, create custom completers, define key bindings, and other settings. These are the kinds of customizations that you want to have available in every PowerShell interactive session. The profile script is the place for these settings.

The simplest way to edit your profile script is to open the file in your favorite code editor. For example, the following command opens the profile in [VS Code](#).

```
PowerShell
```

```
code $PROFILE
```

You could also use `notepad.exe` on Windows, `vi` on Linux, or any other text editor.

The following profile script has examples for many of the customizations mentioned in the previous articles. You can use any of these settings in your own profile.

```
PowerShell
```

```
## Map PSDrives to other registry hives
if (!(Test-Path HKCR:)) {
    $null = New-PSDrive -Name HKCR -PSProvider Registry -Root HKEY_CLASSES_ROOT
    $null = New-PSDrive -Name HKU -PSProvider Registry -Root HKEY_USERS
}

## Customize the prompt
function prompt {
    $identity = [Security.Principal.WindowsIdentity]::GetCurrent()
    $principal = [Security.Principal.WindowsPrincipal] $identity
    $adminRole = [Security.Principal.WindowsBuiltInRole]::Administrator

    $prefix = if (Test-Path Variable:/PSDebugContext) { '[DBG]: ' } else {
        ''
    }
    if ($principal.IsInRole($adminRole)) {
        $prefix = "[ADMIN]:$prefix"
    }
}
```

```

$body = 'PS ' + $PWD.path
$suffix = $($if ($NestedPromptLevel -ge 1) { '>>' }) + '> '
"${prefix}${body}${suffix}"
}

## Create $PSSStyle if running on a version older than 7.2
## - Add other ANSI color definitions as needed

if ($PSVersionTable.PSVersion.ToString() -lt '7.2.0') {
    # define escape char since ``e`` may not be supported
    $esc = [char]0x1b
    $PSSStyle = [pscustomobject]@{
        Foreground = @{
            Magenta = "${esc}[35m"
            BrightYellow = "${esc}[93m"
        }
        Background = @{
            BrightBlack = "${esc}[100m"
        }
    }
}

## Set PSReadLine options and keybindings
$PSROptions = @{
    ContinuationPrompt = ' '
    Colors = @{
        Operator = $PSSStyle.Foreground.Magenta
        Parameter = $PSSStyle.Foreground.Magenta
        Selection = $PSSStyle.Background.BrightBlack
        InLinePrediction = $PSSStyle.Foreground.BrightYellow +
$PSSStyle.Background.BrightBlack
    }
}
Set-PSReadLineOption @PSROptions
Set-PSReadLineKeyHandler -Chord 'Ctrl+f' -Function ForwardWord
Set-PSReadLineKeyHandler -Chord 'Enter' -Function ValidateAndAcceptLine

## Add argument completer for the dotnet CLI tool
$scriptblock = {
    param($wordToComplete, $commandAst, $cursorPosition)
    dotnet complete --position $cursorPosition $commandAst.ToString() |
    ForEach-Object {
        [System.Management.Automation.CompletionResult]::new($_, $_,
'ParameterValue', $_)
    }
}
Register-ArgumentCompleter -Native -CommandName dotnet -ScriptBlock
$scriptblock

```

This profile script provides examples for the following customization:

- Adds two new **PSDrives** for the other root registry hives.
- Creates a **customized prompt** that changes if you're running in an elevated session.

- Configures **PSReadLine** and adds key binding. The color settings use the **\$PSSStyle** feature to define the ANSI color settings.
- Adds tab completion for the **dotnet CLI** tool. The tool provides parameters to help resolve the command-line arguments. The script block for **Register-ArgumentCompleter** uses that feature to provide the tab completion.

Using PSReadLine key handlers

Article • 03/29/2023

The **PSReadLine** module provides key handlers that map **PSReadLine** functions to keyboard chords. Keyboard chords are a sequence of one or more keystrokes that are pressed at the same time. For example, the chord `Ctrl + Spacebar` is the combination of the `Ctrl` and `Spacebar` keys pressed at the same time. A **PSReadLine** function is a predefined action that can be performed on a command line. For example, the `MenuComplete` function allows you to choose from a list of options from a menu complete the input on the command line.

PSReadLine has several predefined key handlers that are bound by default. You can also define your own custom key handlers. Run the following command to list the key handlers that are currently defined.

```
PowerShell
```

```
Get-PSReadLineKeyHandler
```

You can also get a list of all unbound **PSReadLine** functions that are available to be bound to a key chord.

```
PowerShell
```

```
Get-PSReadLineKeyHandler -Unbound
```

You can use the `Set-PSReadLineKeyHandler` cmdlet to bind a function to a key handler. The following command binds the `MenuComplete` function to the chord `Ctrl + Spacebar`.

```
PowerShell
```

```
Set-PSReadLineKeyHandler -Chord 'Ctrl+Spacebar' -Function MenuComplete
```

Finding key names and chord bindings

The names of the keys in the chord are defined by the `[System.ConsoleKey]` enumeration. For more information, see [System.ConsoleKey](#) documentation. For example, the name of the `2` key in `[System.ConsoleKey]` is `D2`, whereas the name of the `2` key on the numeric keypad is `NumPad2`. You can use the `[System.Console]::ReadKey()` method to find the name of the key you pressed.

PowerShell

```
[System.Console]::ReadKey()
```

The following output shows the information returned by the `ReadKey()` method for the `Ctrl+2` key chord.

Output

KeyChar	Key	Modifiers
	D2	Control

For the **PSReadLine** key handler cmdlets, this chord is represented as `Ctrl+D2`. The following example binds this chord to a function.

PowerShell

```
Set-PSReadLineKeyHandler -Chord 'Ctrl+D2' -Function MenuComplete
```

You can bind multiple chords to a single function. By default, the `BackwardDeleteChar` function is bound to two chords.

PowerShell

```
Get-PSReadLineKeyHandler -Chord Backspace, Ctrl+h
```

Output

Key	Function	Description
Backspace	BackwardDeleteChar	Delete the character before the cursor
Ctrl+h	BackwardDeleteChar	Delete the character before the cursor

① Note

The **Chord** parameter is **case-sensitive**. Meaning, you can create different bindings for `Ctrl+x` and `Ctrl+X`.

On Windows, you can also use the `Alt+?` key chord to show the function bound to the next key chord you enter. When you type `Alt+?` you see the following prompt:

Output

what-is-key:

When you hit the `Backspace` key you get the following response:

Output

Backspace: BackwardDeleteChar - Delete the character before the cursor

Key handlers on non-Windows computers

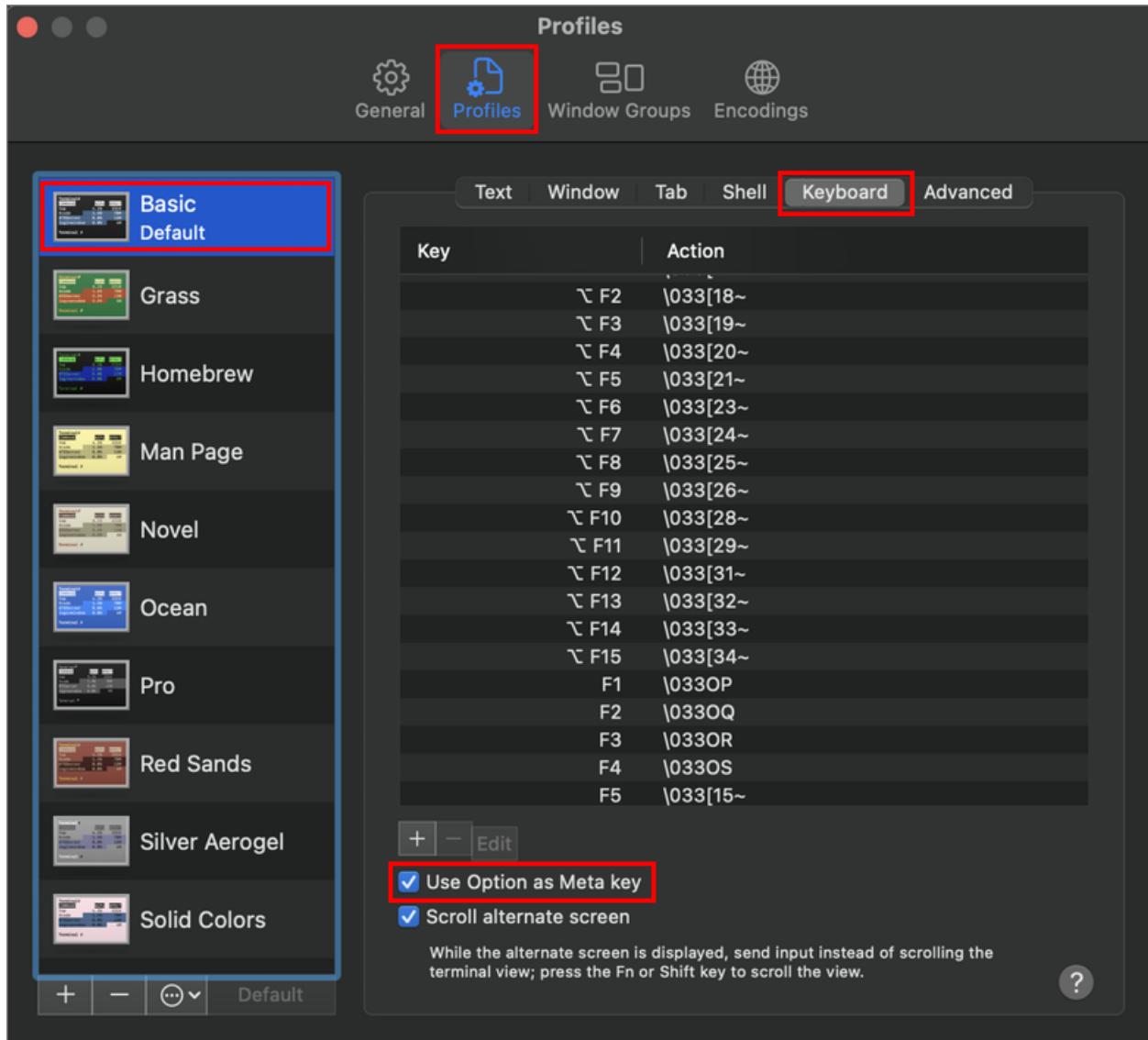
The key codes generated by your keyboard can be different depending on the operating system and terminal application you are using.

macOS

The Macintosh keyboard doesn't have an `Alt` key like Windows and Linux systems. Instead, it has the `⌥ Option` key. macOS uses this key differently than the `Alt` key on other systems. However, you can configure the terminal and iTerm2 applications on macOS to treat it as an `Alt` key.

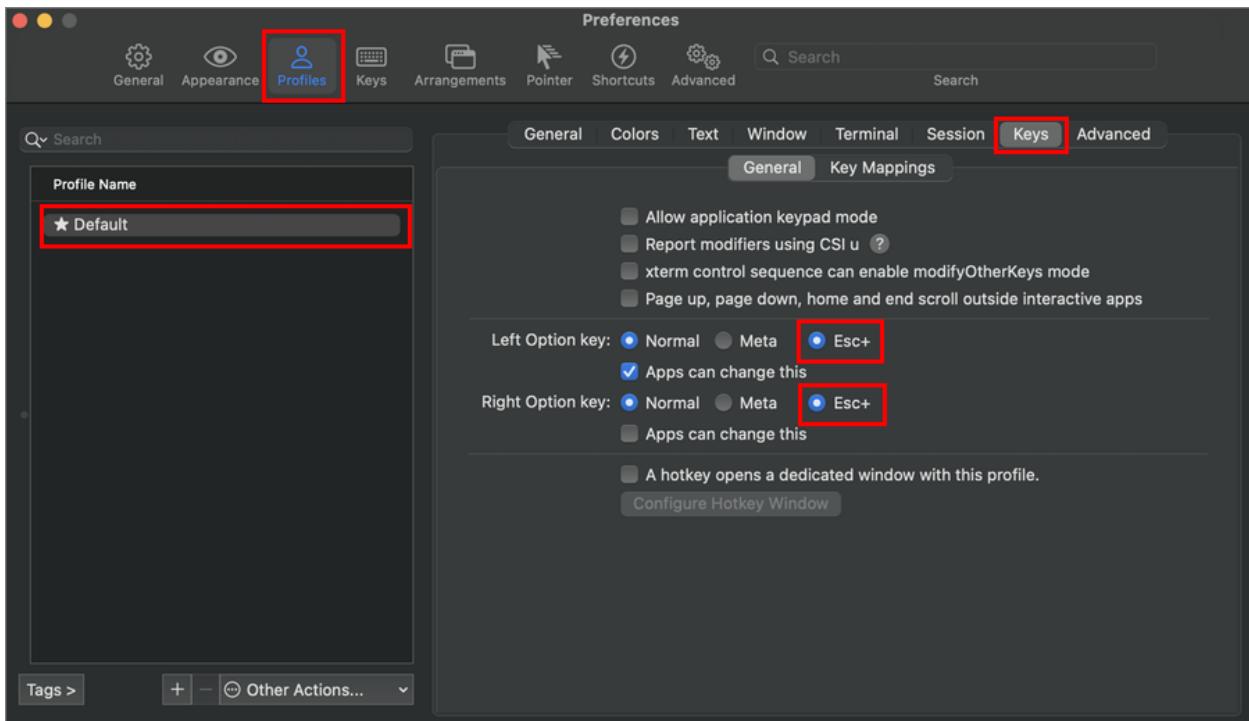
Configuring the Terminal application

Open the **Settings** window from the App bar in Terminal.app. Select **Profiles** and choose the profile you want to configure. Select the **Keyboard** tab of the configuration options. Below the list of keys, select the **Use Option as Meta Key** setting. This setting allows the `⌥ Option` key to act as `Alt` in the Terminal application.



Configuring the iTerm2 application

Open the **Settings** window from the App Bar in iTerm.app. Select **Profiles** and choose the profile you want to configure. Select the **Keys** tab of the configuration options. Select the **Esc+ option** for both the **Left Option Key** and **Right Option Key** settings. This setting allows the **Option** key to act as **Alt** in the iTerm application.



ⓘ Note

The exact steps may vary depending on the versions of macOS and the terminal applications. These examples were captured on macOS Ventura 13.2.1 and iTerm2 v3.4.16.

Linux

On Linux platforms, the key code generated can be different than other systems. For example:

- `Ctrl`+`[` is the same as `Escape`
- `Ctrl`+`Spacebar` generates the key codes for `Ctrl`+`D2`. If you want to map a function `Ctrl`+`Spacebar` you must use the chord `Ctrl+D2`.

PowerShell

```
Set-PSReadLineKeyHandler -Chord 'Ctrl+D2' -Function MenuComplete
```

Use the `.ReadKey()` method to verify the key codes generated by your keyboard.

Commonly used key handlers

Here are a few commonly used key handlers that are bound by default on Windows. Note that the key binding may be different on non-Windows platforms.

MenuComplete

Complete the input by selecting from a menu of possible completion values.

Default chord: `Ctrl+Spacebar`

The following example shows the menu of possible completions for commands beginning with `select`.

```
Output

PS C:\> select<Ctrl+Spacebar>
select          Select-Object          Select-PSFPropertyValue
Select-Xml
Select-AzContext   Select-PSFConfig      Select-PSMDBuildProject
Select-AzSubscription  Select-PSFObject     Select-String

Select-Object
```

Use the arrow keys to select the completion you want. Press the `Enter` key to complete the input. As you move through the selections, help for the selected command is displayed below the menu.

ClearScreen

This function clears the screen similar to the `cls` or `clear` commands.

Default chord: `Ctrl+l`

SelectCommandArgument

Selects the next argument on the command line.

Default chord: `Alt+a`

You may have command in your history that you want to run again with different parameter values. You can use the chord to cycle through each parameter and change the value as needed.

```
New-AzVM -ResourceGroupName myRGName -Location eastus -Name myVM
```

Pressing `Alt+a` selects the next parameter argument in turn: `myRGName`, `eastus`, `myVM`.

GotoBrace

Moves the cursor to the matching brace.

Default chord: `Ctrl+]`

This function moves your cursor to the closing brace that matches the brace at the current cursor position on the command line. The function works for brackets (`[]`), braces (`{}`), and parentheses, (`()`).

DigitArgument

Start or accumulate a numeric argument used to repeat a keystroke the specified number of times.

Default chord: `Alt+0` through `Alt+9`

For example, typing `Alt + 4 + #` enters `####` on the command line.

See also

- [Get-PSReadLineKeyHandler](#)
- [Set-PSReadLineKeyHandler](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

- [Open a documentation issue](#)
- [Provide product feedback](#)

Configuring a light colored theme

Article • 03/07/2023

The default colors for both PowerShell and **PSReadLine** are selected for a dark background terminal. However, some users may choose to use a light background with dark text. Since most of the default colors don't set the background, using light foreground colors on a light background produces unreadable text.

PSReadLine allows you to define colors for 18 different syntax elements. You can view the current settings using the `Get-PSReadLineOption` cmdlet.

Output

```
EditMode : Windows
AddToHistoryHandler :
System.Func`2[System.String,System.Object]
HistoryNoDuplicates : True
HistorySavePath :
C:\Users\user1\AppData\Roaming\Microsoft\Wind...
HistorySaveStyle : SaveIncrementally
HistorySearchCaseSensitive : False
HistorySearchCursorMovesToEnd : False
MaximumHistoryCount : 4096
ContinuationPrompt : >>
ExtraPromptLineCount : 0
PromptText : {> }
BellStyle : Audible
DingDuration : 50
DingTone : 1221
CommandsToValidateScriptBlockArguments : {ForEach-Object, %, Invoke-Command, icm...}
CommandValidationHandler :
CompletionQueryItems : 100
MaximumKillRingCount : 10
ShowToolTips : True
ViModeIndicator : None
WordDelimiters : ;:,.[]{}()\/|^\&*-+='"--_
AnsiEscapeTimeout : 100
PredictionSource : HistoryAndPlugin
PredictionViewStyle : InlineView
CommandColor : ``e[93m"
CommentColor : ``e[32m"
ContinuationPromptColor : ``e[37m"
DefaultTokenColor : ``e[37m"
EmphasisColor : ``e[96m"
ErrorColor : ``e[91m"
InlinePredictionColor : ``e[38;5;238m"
KeywordColor : ``e[92m"
ListPredictionColor : ``e[33m"
ListPredictionSelectedColor : ``e[48;5;238m"
```

```

MemberColor           : ``e[97m"
NumberColor          : ``e[97m"
OperatorColor        : ``e[90m"
ParameterColor       : ``e[90m"
SelectionColor       : ``e[30;47m"
StringColor          : ``e[36m"
TypeColor            : ``e[37m"
VariableColor        : ``e[92m"

```

The color settings are stored as strings containing ANSI escape sequences that change the color in your terminal. Using the `Set-PSReadLineOption` cmdlet you can change the colors to values that work better for a light-colored background.

Defining colors for a light theme

The PowerShell ISE can be configured to use a light theme for both the editor and console panes. You can also view and change the colors that the ISE uses for various syntax and output types. You can use these color choices to define a similar theme for `PSReadLine`.

The following hashtable defines colors for `PSReadLine` that mimic the colors in the PowerShell ISE.

PowerShell

```

$ISETheme = @{
    Command           = $PSStyle.Foreground.FromRGB(0x0000FF)
    Comment          = $PSStyle.Foreground.FromRGB(0x006400)
    ContinuationPrompt = $PSStyle.Foreground.FromRGB(0x0000FF)
    Default          = $PSStyle.Foreground.FromRGB(0x0000FF)
    Emphasis          = $PSStyle.Foreground.FromRGB(0x287BF0)
    Error             = $PSStyle.Foreground.FromRGB(0xE50000)
    InlinePrediction = $PSStyle.Foreground.FromRGB(0x93A1A1)
    Keyword           = $PSStyle.Foreground.FromRGB(0x00008b)
    ListPrediction   = $PSStyle.Foreground.FromRGB(0x06DE00)
    Member            = $PSStyle.Foreground.FromRGB(0x000000)
    Number            = $PSStyle.Foreground.FromRGB(0x800080)
    Operator           = $PSStyle.Foreground.FromRGB(0x757575)
    Parameter          = $PSStyle.Foreground.FromRGB(0x000080)
    String             = $PSStyle.Foreground.FromRGB(0x8b0000)
    Type               = $PSStyle.Foreground.FromRGB(0x008080)
    Variable           = $PSStyle.Foreground.FromRGB(0xff4500)
    ListPredictionSelected = $PSStyle.Background.FromRGB(0x93A1A1)
    Selection          = $PSStyle.Background.FromRGB(0x00BFFF)
}

```

 Note

In PowerShell 7.2 and higher you can use the `FromRGB()` method of `$PSSStyle` to create the ANSI escape sequences for the colors you want.

For more information about `$PSSStyle`, see [about_ANSI_Terminals](#).

For more information about ANSI escape sequences, see the [ANSI escape code](#) article in Wikipedia.

Setting the color theme in your profile

To have the color settings you want in every PowerShell session, you must add the configuration settings to your PowerShell profile script. For an example, see [Customizing your shell environment](#)

Add the `$ISETheme` variable and the following `Set-PSReadLineOption` command to your profile.

```
PowerShell
```

```
Set-PSReadLineOption -Colors $ISETheme
```

Beginning in PowerShell 7.2, PowerShell adds colorized output to the default console experience. The colors used are defined in the `$PSSStyle` variable and are designed for a dark background. The following settings work better for a light background terminal.

```
PowerShell
```

```
$PSSStyle.Formatting.FormatAccent      = ``e[32m"
$PSSStyle.Formatting.TableHeader       = ``e[32m"
$PSSStyle.Formatting.ErrorAccent       = ``e[36m"
$PSSStyle.Formatting.Error            = ``e[31m"
$PSSStyle.Formatting.Warning          = ``e[33m"
$PSSStyle.Formatting.Verbose          = ``e[33m"
$PSSStyle.Formatting.Debug             = ``e[33m"
$PSSStyle.Progress.Style              = ``e[33m"
$PSSStyle.FileInfo.Directory          =
$PSSStyle.Background.FromRgb(0x2f6aff) +
                                         $PSSStyle.Foreground.BrightWhite
$PSSStyle.FileInfo.SymbolicLink        = ``e[36m"
$PSSStyle.FileInfo.Executable          = ``e[95m"
$PSSStyle.FileInfo.Extension['.ps1']   = ``e[36m"
$PSSStyle.FileInfo.Extension['.ps1xml'] = ``e[36m"
$PSSStyle.FileInfo.Extension['.psd1']   = ``e[36m"
$PSSStyle.FileInfo.Extension['.psm1']   = ``e[36m"
```

Choosing colors for accessibility

The ISE color theme may not work for users with color-blindness or other conditions that limit their ability to see colors.

The [World Wide Web Consortium \(W3C\)](#) has recommendations for using colors for accessibility. The Web Content Accessibility Guidelines (WCAG) 2.1 recommends that "visual presentation of text and images of text has a contrast ratio of at least 4.5:1." For more information, see [Success Criterion 1.4.3 Contrast \(Minimum\)](#).

The [Contrast Ratio](#) website provides a tool that lets you pick foreground and background colors and measure the contrast. You can use this tool to find color combinations that work best for you.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Improve the accessibility of output in PowerShell

Article • 09/12/2024

Most terminal environments only display raw text. Users that rely on screen readers are faced with tedious narration when consuming large amounts of raw text because the raw output doesn't have the accessibility metadata to characterize the format of the content.

There are two ways to improve the accessibility of the output in PowerShell:

- Output the data in a way that it can be viewed in another tool that supports screen reading technologies.
- Reduce the amount of output displayed in the terminal by filtering and selecting the data you want and output the text in a more readable format.

Display the data in a tool outside of the terminal

For large amounts of data, rather than output to the host, consider writing output in a format that can be viewed in another tool that supports screen reading technologies. You might need to save the data to a file in a format that can be opened in another application.

Out-GridView command on Windows

For small to moderate size output, use the `Out-GridView` command. The output is rendered using Windows Presentation Foundation (WPF) in tabular form, much like a spreadsheet. The GridView control allows you to sort, filter, and search the data, which reduces the amount of data that needs to be read. The GridView control is also accessible to screen readers. The **Narrator** tool built into Windows is able to read the GridView details, including column names and row count.

The following example shows how to display a list of services in a GridView control.

PowerShell

```
Get-Service | Out-GridView
```

The `Out-GridView` command is only available in PowerShell on Windows.

Character Separated Value (CSV) format

Spreadsheet applications such as **Microsoft Excel** support CSV files. The following example shows how to save the output of a command to a CSV file.

PowerShell

```
Get-Service | Export-Csv -Path .\myFile.csv  
Invoke-Item .\myFile.csv
```

The `Invoke-Item` command opens the file in the default application for CSV files, which is usually Microsoft Excel.

HyperText Markup Language (HTML) format

HTML files can be viewed by web browsers such as **Microsoft Edge**. The following example shows how to save the output of a command to an HTML file.

PowerShell

```
Get-Service | ConvertTo-HTML | Out-File .\myFile.html  
Invoke-Item .\myFile.html
```

The `Invoke-Item` command opens the file in your default web browser.

Reduce the amount of output

One way to improve the accessibility of the output is to reduce the amount of output displayed in the terminal. PowerShell has several commands that can help you filter and select the data you want.

Select and filter data

Rather than returning a large mount of data, use commands such as `Select-Object`, `Sort-Object`, and `Where-Object` to reduce the amount of output. The following example gets the list of services on the computer.

Each of the following commands improves the output in a different way:

- The `-ErrorAction SilentlyContinue` parameter suppresses error messages that might be generated if the user doesn't have permission to view some services.
- The `Where-Object` command reduces the number of items returned by filtering the list to only show services that are running and have `event` in the description.
- The `Select-Object` command selects only the service name and display name.
- The `Format-List` command displays the output in list format, which provides a better narration experience for screen readers.

PowerShell

```
Get-Service -ErrorAction SilentlyContinue |
    Where-Object {$_.Status -eq 'Running' -and $_.Description -match
    'event'} |
        Select-Object Name, DisplayName |
            Format-List
```

Reformat the output with calculated properties

The default property names of .NET objects output by PowerShell can be verbose and confusing. You can use calculated properties to change the property names and values to something easier to understand when read by a narrator technology.

The following example shows how to get the top five processes by memory usage and display the process name and memory usage in megabytes.

PowerShell

```
Get-Process |
    Sort-Object WorkingSet -Descending |
        Select-Object -First 5 -Property ProcessName,
            @{n="MemoryMB"; e={'{0:N2}' -f ($_.WorkingSet/1Mb)}} |
                Format-List
```

By default, `Get-Process` displays the `WorkingSet` as the number of bytes of memory used. Without formatting, it can be difficult to understand the magnitude of the number. The calculated property converts the number of bytes to megabytes and formats the number with commas and limits the value to two decimal places.

Output

```
ProcessName : vmmemWSL
MemoryMB   : 1,217.69

ProcessName : Memory Compression
MemoryMB   : 780.45
```

```
ProcessName : Code  
MemoryMB    : 726.43
```

```
ProcessName : OUTLOOK  
MemoryMB   : 460.16
```

```
ProcessName : msedgewebview2  
MemoryMB   : 428.94
```

Additional reading

- [Out-GridView](#)
- [Export-Csv](#)
- [ConvertTo-Html](#)
- [about_Calculated_Properties](#)

Deep dive articles

Article • 11/17/2022

The articles in this section are designed to be an in-depth look into PowerShell topics. These articles don't replace the reference articles, but provide diverse examples, illustrate edge cases, and warn about pitfalls and common mistakes.

This collection is also a showcase for community contributions. The inaugural set of articles come from [@KevinMarquette](#) and were originally published at [PowerShellExplained.com](#).

How to contribute content

If you're interested in contributing content to this collection, please read the [Contributor Guide](#). When you are ready to propose a contribution, submit an issue in the GitHub repository using the [Document Idea template](#) and include a link to the existing content you want to share.



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Everything you wanted to know about arrays

Article • 06/20/2024

Arrays are a fundamental language feature of most programming languages. They're a collection of values or objects that are difficult to avoid. Let's take a close look at arrays and everything they have to offer.

ⓘ Note

The [original version](#) of this article appeared on the blog written by [@KevinMarquette](#). The PowerShell team thanks Kevin for sharing this content with us. Please check out his blog at [PowerShellExplained.com](#).

What is an array?

I'm going to start with a basic technical description of what arrays are and how they are used by most programming languages before I shift into the other ways PowerShell makes use of them.

An array is a data structure that serves as a collection of multiple items. You can iterate over the array or access individual items using an index. The array is created as a sequential chunk of memory where each value is stored right next to the other.

I'll touch on each of those details as we go.

Basic usage

Because arrays are such a basic feature of PowerShell, there is a simple syntax for working with them in PowerShell.

Create an array

An empty array can be created by using `@()`

PowerShell

```
PS> $data = @()
PS> $data.Count
```

We can create an array and seed it with values just by placing them in the `@()` parentheses.

PowerShell

```
PS> $data = @('Zero', 'One', 'Two', 'Three')
PS> $data.Count
4

PS> $data
Zero
One
Two
Three
```

This array has 4 items. When we call the `$data` variable, we see the list of our items. If it's an array of strings, then we get one line per string.

We can declare an array on multiple lines. The comma is optional in this case and generally left out.

PowerShell

```
$data = @(
    'Zero'
    'One'
    'Two'
    'Three'
)
```

I prefer to declare my arrays on multiple lines like that. Not only does it get easier to read when you have multiple items, it also makes it easier to compare to previous versions when using source control.

Other syntax

It's commonly understood that `@()` is the syntax for creating an array, but comma-separated lists work most of the time.

PowerShell

```
$data = 'Zero', 'One', 'Two', 'Three'
```

Write-Output to create arrays

One cool little trick worth mentioning is that you can use `Write-Output` to quickly create strings at the console.

```
PowerShell
```

```
$data = Write-Output Zero One Two Three
```

This is handy because you don't have to put quotes around the strings when the parameter accepts strings. I would never do this in a script but it's fair game in the console.

Accessing items

Now that you have an array with items in it, you may want to access and update those items.

Offset

To access individual items, we use the brackets `[]` with an offset value starting at 0. This is how we get the first item in our array:

```
PowerShell
```

```
PS> $data = 'Zero', 'One', 'Two', 'Three'  
PS> $data[0]  
Zero
```

The reason why we use zero here is because the first item is at the beginning of the list so we use an offset of 0 items to get to it. To get to the second item, we would need to use an offset of 1 to skip the first item.

```
PowerShell
```

```
PS> $data[1]  
One
```

This would mean that the last item is at offset 3.

```
PowerShell
```

```
PS> $data[3]
```

```
Three
```

Index

Now you can see why I picked the values that I did for this example. I introduced this as an offset because that is what it really is, but this offset is more commonly referred to as an index. An index that starts at `0`. For the rest of this article I will call the offset an index.

Special index tricks

In most languages, you can only specify a single number as the index and you get a single item back. PowerShell is much more flexible. You can use multiple indexes at once. By providing a list of indexes, we can select several items.

```
PowerShell
```

```
PS> $data[0,2,3]
```

```
Zero
```

```
Two
```

```
Three
```

The items are returned based on the order of the indexes provided. If you duplicate an index, you get that item both times.

```
PowerShell
```

```
PS> $data[3,0,3]
```

```
Three
```

```
Zero
```

```
Three
```

We can specify a sequence of numbers with the built-in `..` operator.

```
PowerShell
```

```
PS> $data[1..3]
```

```
One
```

```
Two
```

```
Three
```

This works in reverse too.

PowerShell

```
PS> $data[3..1]
Three
Two
One
```

You can use negative index values to offset from the end. So if you need the last item in the list, you can use `-1`.

PowerShell

```
PS> $data[-1]
Three
```

One word of caution here with the `..` operator. The sequence `0..-1` and `-1..0` evaluate to the values `0,-1` and `-1,0`. It's easy to see `$data[0..-1]` and think it would enumerate all items if you forget this detail. `$data[0..-1]` gives you the same value as `$data[0,-1]` by giving you the first and last item in the array (and none of the other values). Here is a larger example:

PowerShell

```
PS> $a = 1,2,3,4,5,6,7,8
PS> $a[2..-1]
3
2
1
8
```

This is the same as:

PowerShell

```
PS> $a[2,1,0,-1]
3
2
1
8
```

Out of bounds

In most languages, if you try to access an index of an item that is past the end of the array, you would get some type of error or an exception. PowerShell silently returns

nothing.

```
PowerShell
```

```
PS> $null -eq $data[9000]
True
```

Cannot index into a null array

If your variable is `$null` and you try to index it like an array, you get a `System.Management.Automation.RuntimeException` exception with the message `Cannot index into a null array.`.

```
PowerShell
```

```
PS> $empty = $null
PS> $empty[0]
Error: Cannot index into a null array.
```

So make sure your arrays are not `$null` before you try to access elements inside them.

Count

Arrays and other collections have a `Count` property that tells you how many items are in the array.

```
PowerShell
```

```
PS> $data.Count
4
```

PowerShell 3.0 added a `Count` property to most objects. you can have a single object and it should give you a count of `1`.

```
PowerShell
```

```
PS> $date = Get-Date
PS> $date.Count
1
```

Even `$null` has a `Count` property except it returns `0`.

```
PowerShell
```

```
PS> $null.Count  
0
```

There are some traps here that I will revisit when I cover checking for `$null` or empty arrays later on in this article.

Off-by-one errors

A common programming error is created because arrays start at index 0. Off-by-one errors can be introduced in two ways.

The first is by mentally thinking you want the second item and using an index of `2` and really getting the third item. Or by thinking that you have four items and you want last item, so you use the count to access the last item.

```
PowerShell
```

```
$data[ $data.Count ]
```

PowerShell is perfectly happy to let you do that and give you exactly what item exists at index 4: `$null`. You should be using `$data.Count - 1` or the `-1` that we learned about above.

```
PowerShell
```

```
PS> $data[ $data.Count - 1 ]  
Three
```

This is where you can use the `-1` index to get the last element.

```
PowerShell
```

```
PS> $data[ -1 ]  
Three
```

Lee Dailey also pointed out to me that we can use `$data.GetUpperBound(0)` to get the max index number.

```
PowerShell
```

```
PS> $data.GetUpperBound(0)  
3
```

```
PS> $data[ $data.GetUpperBound(0) ]  
Three
```

The second most common way is when iterating the list and not stopping at the right time. I'll revisit this when we talk about using the `for` loop.

Updating items

We can use the same index to update existing items in the array. This gives us direct access to update individual items.

```
PowerShell
```

```
$data[2] = 'dos'  
$data[3] = 'tres'
```

If we try to update an item that is past the last element, then we get an `Index was outside the bounds of the array.` error.

```
PowerShell
```

```
PS> $data[4] = 'four'  
Index was outside the bounds of the array.  
At line:1 char:1  
+ $data[4] = 'four'  
+ ~~~~~~  
+ CategoryInfo          : OperationStopped: () [], IndexOutOfRangeException  
+ FullyQualifiedErrorId : System.IndexOutOfRangeException
```

I'll revisit this later when I talk about how to make an array larger.

Iteration

At some point, you might need to walk or iterate the entire list and perform some action for each item in the array.

Pipeline

Arrays and the PowerShell pipeline are meant for each other. This is one of the simplest ways to process over those values. When you pass an array to a pipeline, each item inside the array is processed individually.

```
PowerShell
```

```
PS> $data = 'Zero', 'One', 'Two', 'Three'  
PS> $data | ForEach-Object {"Item: [$PSItem]"}  
Item: [Zero]  
Item: [One]  
Item: [Two]  
Item: [Three]
```

If you have not seen `$PSItem` before, just know that it's the same thing as `$_.` You can use either one because they both represent the current object in the pipeline.

ForEach loop

The `foreach` loop works well with collections. Using the syntax: `foreach (<variable> in <collection>)`

PowerShell

```
foreach ( $node in $data )  
{  
    "Item: [$node]"  
}
```

ForEach method

I tend to forget about this one but it works well for simple operations. PowerShell allows you to call `ForEach()` on a collection.

PowerShell

```
PS> $data.ForEach({"Item: [$PSItem]"}  
Item: [Zero]  
Item: [One]  
Item: [Two]  
Item: [Three]
```

The `ForEach()` takes a parameter that is a script block. You can drop the parentheses and just provide the script block.

PowerShell

```
$data.ForEach{"Item: [$PSItem]"}  
Item: [Zero]  
Item: [One]  
Item: [Two]  
Item: [Three]
```

This is a lesser known syntax but it works just the same. This `ForEach` method was added in PowerShell 4.0.

For loop

The `for` loop is used heavily in most other languages but you don't see it much in PowerShell. When you do see it, it's often in the context of walking an array.

PowerShell

```
for ( $index = 0; $index -lt $data.Count; $index++)
{
    "Item: [{0}]" -f $data[$index]
}
```

The first thing we do is initialize an `$index` to `0`. Then we add the condition that `$index` must be less than `$data.Count`. Finally, we specify that every time we loop that we must increase the index by `1`. In this case `$index++` is short for `$index = $index + 1`. The [format operator](#) (`-f`) is used to insert the value of `$data[$index]` in the output string.

Whenever you're using a `for` loop, pay special attention to the condition. I used `$index -lt $data.Count` here. It's easy to get the condition slightly wrong to get an off-by-one error in your logic. Using `$index -le $data.Count` or `$index -lt ($data.Count - 1)` are ever so slightly wrong. That would cause your result to process too many or too few items. This is the classic off-by-one error.

Switch loop

This is one that is easy to overlook. If you provide an array to a [switch statement](#), it checks each item in the array.

PowerShell

```
$data = 'Zero', 'One', 'Two', 'Three'
switch( $data )
{
    'One'
    {
        'Tock'
    }
    'Three'
    {
        'Tock'
    }
    Default
```

```
{  
    'Tick'  
}  
}
```

Output

```
Tick  
Tock  
Tick  
Tock
```

There are a lot of cool things that we can do with the switch statement. I have another article dedicated to this.

- [Everything you ever wanted to know about the switch statement](#)

Updating values

When your array is a collection of string or integers (value types), sometimes you may want to update the values in the array as you loop over them. Most of the loops above use a variable in the loop that holds a copy of the value. If you update that variable, the original value in the array is not updated.

The exception to that statement is the `for` loop. If you want to walk an array and update values inside it, then the `for` loop is what you're looking for.

PowerShell

```
for ( $index = 0; $index -lt $data.Count; $index++ )  
{  
    $data[$index] = "Item: [{0}]" -f $data[$index]  
}
```

This example takes a value by index, makes a few changes, and then uses that same index to assign it back.

Arrays of Objects

So far, the only thing we've placed in an array is a value type, but arrays can also contain objects.

PowerShell

```
$data = @(
    [pscustomobject]@{FirstName='Kevin';LastName='Marquette'}
    [pscustomobject]@{FirstName='John'; LastName='Doe'}
)
```

Many cmdlets return collections of objects as arrays when you assign them to a variable.

PowerShell

```
$processList = Get-Process
```

All of the basic features we already talked about still apply to arrays of objects with a few details worth pointing out.

Accessing properties

We can use an index to access an individual item in a collection just like with value types.

PowerShell

```
PS> $data[0]

FirstName LastName
----- -----
Kevin      Marquette
```

We can access and update properties directly.

PowerShell

```
PS> $data[0].FirstName

Kevin

PS> $data[0].FirstName = 'Jay'
PS> $data[0]

FirstName LastName
----- -----
Jay       Marquette
```

Array properties

Normally you would have to enumerate the whole list like this to access all the properties:

```
PowerShell

PS> $data | ForEach-Object {$_.LastName}

Marquette
Doe
```

Or by using the `Select-Object -ExpandProperty` cmdlet.

```
PowerShell

PS> $data | Select-Object -ExpandProperty LastName

Marquette
Doe
```

But PowerShell offers us the ability to request `LastName` directly. PowerShell enumerates them all for us and returns a clean list.

```
PowerShell

PS> $data.LastName

Marquette
Doe
```

The enumeration still happens but we don't see the complexity behind it.

Where-Object filtering

This is where `Where-Object` comes in so we can filter and select what we want out of the array based on the properties of the object.

```
PowerShell

PS> $data | Where-Object {$_.FirstName -eq 'Kevin'}

FirstName LastName
----- -----
Kevin      Marquette
```

We can write that same query to get the `FirstName` we are looking for.

```
PowerShell
```

```
$data | where FirstName -EQ Kevin
```

Where()

Arrays have a `Where()` method on them that allows you to specify a `scriptblock` for the filter.

```
PowerShell
```

```
$data.Where({$_.FirstName -eq 'Kevin'})
```

This feature was added in PowerShell 4.0.

Updating objects in loops

With value types, the only way to update the array is to use a for loop because we need to know the index to replace the value. We have more options with objects because they are reference types. Here is a quick example:

```
PowerShell
```

```
foreach($person in $data)
{
    $person.FirstName = 'Kevin'
}
```

This loop is walking every object in the `$data` array. Because objects are reference types, the `$person` variable references the exact same object that is in the array. So updates to its properties do update the original.

You still can't replace the whole object this way. If you try to assign a new object to the `$person` variable, you're updating the variable reference to something else that no longer points to the original object in the array. This doesn't work like you would expect:

```
PowerShell
```

```
foreach($person in $data)
{
    $person = [pscustomobject]@{
        FirstName='Kevin'
        LastName='Marquette'
```

```
}
```

Operators

The operators in PowerShell also work on arrays. Some of them work slightly differently.

-join

The `-join` operator is the most obvious one so let's look at it first. I like the `-join` operator and use it often. It joins all elements in the array with the character or string that you specify.

PowerShell

```
PS> $data = @(1,2,3,4)
PS> $data -join '-'
1-2-3-4
PS> $data -join ','
1,2,3,4
```

One of the features that I like about the `-join` operator is that it handles single items.

PowerShell

```
PS> 1 -join '-'
1
```

I use this inside logging and verbose messages.

PowerShell

```
PS> $data = @(1,2,3,4)
PS> "Data is $($data -join ',')."
Data is 1,2,3,4.
```

-join \$array

Here is a clever trick that Lee Dailey pointed out to me. If you ever want to join everything without a delimiter, instead of doing this:

PowerShell

```
PS> $data = @(1,2,3,4)
PS> $data -join $null
1234
```

You can use `-join` with the array as the parameter with no prefix. Take a look at this example to see that I'm talking about.

PowerShell

```
PS> $data = @(1,2,3,4)
PS> -join $data
1234
```

-replace and -split

The other operators like `-replace` and `-split` execute on each item in the array. I can't say that I have ever used them this way but here is an example.

PowerShell

```
PS> $data = @('ATX-SQL-01','ATX-SQL-02','ATX-SQL-03')
PS> $data -replace 'ATX','LAX'
LAX-SQL-01
LAX-SQL-02
LAX-SQL-03
```

-contains

The `-contains` operator allows you to check an array of values to see if it contains a specified value.

PowerShell

```
PS> $data = @('red','green','blue')
PS> $data -contains 'green'
True
```

-in

When you have a single value that you would like to verify matches one of several values, you can use the `-in` operator. The value would be on the left and the array on the right-hand side of the operator.

PowerShell

```
PS> $data = @('red','green','blue')
PS> 'green' -in $data
True
```

This can get expensive if the list is large. I often use a regex pattern if I'm checking more than a few values.

PowerShell

```
PS> $data = @('red','green','blue')
PS> $pattern = "^({0})$" -f ($data -join '|')
PS> $pattern
^(red|green|blue)$

PS> 'green' -match $pattern
True
```

-eq and -ne

Equality and arrays can get complicated. When the array is on the left side, every item gets compared. Instead of returning `True`, it returns the object that matches.

PowerShell

```
PS> $data = @('red','green','blue')
PS> $data -eq 'green'
green
```

When you use the `-ne` operator, we get all the values that are not equal to our value.

PowerShell

```
PS> $data = @('red','green','blue')
PS> $data -ne 'green'
red
blue
```

When you use this in an `if()` statement, a value that is returned is a `True` value. If no value is returned, then it's a `False` value. Both of these next statements evaluate to `True`.

PowerShell

```
$data = @('red','green','blue')
if ( $data -eq 'green' )
{
    'Green was found'
}
if ( $data -ne 'green' )
{
    'And green was not found'
}
```

I'll revisit this in a moment when we talk about testing for `$null`.

-match

The `-match` operator tries to match each item in the collection.

PowerShell

```
PS> $servers = @(
    'LAX-SQL-01'
    'LAX-API-01'
    'ATX-SQL-01'
    'ATX-API-01'
)
PS> $servers -match 'SQL'
LAX-SQL-01
ATX-SQL-01
```

When you use `-match` with a single value, a special variable `$Matches` gets populated with match info. This isn't the case when an array is processed this way.

We can take the same approach with `Select-String`.

PowerShell

```
$servers | Select-String SQL
```

I take a closer look at `Select-String`, `-match` and the `$Matches` variable in another post called [The many ways to use regex ↴](#).

\$null or empty

Testing for `$null` or empty arrays can be tricky. Here are the common traps with arrays.

At a glance, this statement looks like it should work.

PowerShell

```
if ( $array -eq $null)
{
    'Array is $null'
}
```

But I just went over how `-eq` checks each item in the array. So we can have an array of several items with a single `$null` value and it would evaluate to `$true`

PowerShell

```
$array = @('one',$null,'three')
if ( $array -eq $null)
{
    'I think Array is $null, but I would be wrong'
}
```

This is why it's a best practice to place the `$null` on the left side of the operator. This makes this scenario a non-issue.

PowerShell

```
if ( $null -eq $array )
{
    'Array actually is $null'
}
```

A `$null` array isn't the same thing as an empty array. If you know you have an array, check the count of objects in it. If the array is `$null`, the count is `0`.

PowerShell

```
if ( $array.Count -gt 0 )
{
    "Array isn't empty"
}
```

There is one more trap to watch out for here. You can use the `Count` even if you have a single object, unless that object is a `PSCustomObject`. This is a bug that is fixed in PowerShell 6.1. That's good news, but a lot of people are still on 5.1 and need to watch out for it.

PowerShell

```
PS> $object = [pscustomobject]@{Name='TestObject'}
PS> $object.Count
$null
```

If you're still on PowerShell 5.1, you can wrap the object in an array before checking the count to get an accurate count.

PowerShell

```
if ( @($array).Count -gt 0 )
{
    "Array isn't empty"
}
```

To fully play it safe, check for `$null`, then check the count.

PowerShell

```
if ( $null -ne $array -and @($array).Count -gt 0 )
{
    "Array isn't empty"
}
```

All -eq

I recently saw someone on Reddit ask how to verify that every value in an array matches a given value. Reddit user `u/bis` had this clever solution that checks for any incorrect values and then flips the result.

PowerShell

```
$results = Test-Something
if ( -not ( $results -ne 'Passed' ) )
{
    'All results a Passed'
```

Adding to arrays

At this point, you're starting to wonder how to add items to an array. The quick answer is that you can't. An array is a fixed size in memory. If you need to grow it or add a single item to it, then you need to create a new array and copy all the values over from

the old array. This sounds like a lot of work, however, PowerShell hides the complexity of creating the new array. PowerShell implements the addition operator (+) for arrays.

ⓘ Note

PowerShell does not implement a subtraction operation. If you want a flexible alternative to an array, you need to use a [generic List](#) object.

Array addition

We can use the addition operator with arrays to create a new array. So given these two arrays:

PowerShell

```
$first = @(  
    'Zero'  
    'One'  
)  
$second = @(  
    'Two'  
    'Three'  
)
```

We can add them together to get a new array.

PowerShell

```
PS> $first + $second  
  
Zero  
One  
Two  
Three
```

Plus equals +=

We can create a new array in place and add an item to it like this:

PowerShell

```
$data = @(  
    'Zero'  
    'One'  
    'Two'
```

```
'Three'  
)  
$data += 'four'
```

Just remember that every time you use `+=` that you're duplicating and creating a new array. This is a not an issue for small datasets but it scales extremely poorly.

Pipeline assignment

You can assign the results of any pipeline into a variable. It's an array if it contains multiple items.

PowerShell

```
$array = 1..5 | ForEach-Object {  
    "ATX-SQL-$PSItem"  
}
```

Normally when we think of using the pipeline, we think of the typical PowerShell one-liners. We can leverage the pipeline with `foreach()` statements and other loops. So instead of adding items to an array in a loop, we can drop items onto the pipeline.

PowerShell

```
$array = foreach ( $node in (1..5) )  
{  
    "ATX-SQL-$node"  
}
```

Array Types

By default, an array in PowerShell is created as a `[psobject[]]` type. This allows it to contain any type of object or value. This works because everything is inherited from the `PSObject` type.

Strongly typed arrays

You can create an array of any type using a similar syntax. When you create a strongly typed array, it can only contain values or objects the specified type.

PowerShell

```
PS> [int[]] $numbers = 1,2,3
PS> [int[]] $numbers2 = 'one','two','three'
ERROR: Cannot convert value "one" to type "System.Int32". Input string was
not in a correct format.

PS> [string[]] $strings = 'one','two','three'
```

ArrayList

Adding items to an array is one of its biggest limitations, but there are a few other collections that we can turn to that solve this problem.

The `ArrayList` is commonly one of the first things that we think of when we need an array that is faster to work with. It acts like an object array every place that we need it, but it handles adding items quickly.

Here is how we create an `ArrayList` and add items to it.

PowerShell

```
$myarray = [System.Collections.ArrayList]::new()
[void]$myArray.Add('Value')
```

We are calling into .NET to get this type. In this case, we are using the default constructor to create it. Then we call the `Add` method to add an item to it.

The reason I'm using `[void]` at the beginning of the line is to suppress the return code. Some .NET calls do this and can create unexpected output.

If the only data that you have in your array is strings, then also take a look at using `StringBuilder`. It's almost the same thing but has some methods that are just for dealing with strings. The `StringBuilder` is specially designed for performance.

It's common to see people move to `ArrayList` from arrays. But it comes from a time where C# didn't have generic support. The `ArrayList` is deprecated in support for the generic `List[]`

Generic List

A generic type is a special type in C# that defines a generalized class and the user specifies the data types it uses when created. So if you want a list of numbers or strings, you would define that you want list of `int` or `string` types.

Here is how you create a List for strings.

```
PowerShell
```

```
$mylist = [System.Collections.Generic.List[string]]::new()
```

Or a list for numbers.

```
PowerShell
```

```
$mylist = [System.Collections.Generic.List[int]]::new()
```

We can cast an existing array to a list like this without creating the object first:

```
PowerShell
```

```
$mylist = [System.Collections.Generic.List[int]]@(1,2,3)
```

We can shorten the syntax with the `using namespace` statement in PowerShell 5 and newer. The `using` statement needs to be the first line of your script. By declaring a namespace, PowerShell lets you leave it off of the data types when you reference them.

```
PowerShell
```

```
using namespace System.Collections.Generic  
$myList = [List[int]]@(1,2,3)
```

This makes the `List` much more usable.

You have a similar `Add` method available to you. Unlike the `ArrayList`, there is no return value on the `Add` method so we don't have to `void` it.

```
PowerShell
```

```
$myList.Add(10)
```

And we can still access the elements like other arrays.

```
PowerShell
```

```
PS> $myList[-1]  
10
```

List[psobject]

You can have a list of any type, but when you don't know the type of objects, you can use `[List[psobject]]` to contain them.

PowerShell

```
$list = [List[psobject]]::new()
```

Remove()

The `ArrayList` and the generic `List[]` both support removing items from the collection.

PowerShell

```
using namespace System.Collections.Generic
$list = [List[string]]@('Zero', 'One', 'Two', 'Three')
[void]$list.Remove("Two")
Zero
One
Three
```

When working with value types, it removes the first one from the list. You can call it over and over again to keep removing that value. If you have reference types, you have to provide the object that you want removed.

PowerShell

```
[List[System.Management.Automation.PSDriveInfo]]$drives = Get-PSDrive
$drives.Remove($drives[2])
```

PowerShell

```
$delete = $drives[2]
$drives.Remove($delete)
```

The remove method returns `true` if it was able to find and remove the item from the collection.

More collections

There are many other collections that can be used but these are the good generic array replacements. If you're interested in learning about more of these options, take a look at this [Gist](#) that [Mark Kraus](#) put together.

Other nuances

Now that I have covered all the major functionality, here are a few more things that I wanted to mention before I wrap this up.

Pre-sized arrays

I mentioned that you can't change the size of an array once it's created. We can create an array of a pre-determined size by calling it with the `new($size)` constructor.

```
PowerShell
```

```
$data = [Object[]]::new(4)  
$data.Count  
4
```

Multiplying arrays

An interesting little trick is that you can multiply an array by an integer.

```
PowerShell
```

```
PS> $data = @('red','green','blue')  
PS> $data * 3  
red  
green  
blue  
red  
green  
blue  
red  
green  
blue
```

Initialize with 0

A common scenario is that you want to create an array with all zeros. If you're only going to have integers, a strongly typed array of integers defaults to all zeros.

PowerShell

```
PS> [int[]]::new(4)
0
0
0
0
```

We can use the multiplying trick to do this too.

PowerShell

```
PS> $data = @(0) * 4
PS> $data
0
0
0
0
```

The nice thing about the multiplying trick is that you can use any value. So if you would rather have 255 as your default value, this would be a good way to do it.

PowerShell

```
PS> $data = @(255) * 4
PS> $data
255
255
255
255
```

Nested arrays

An array inside an array is called a nested array. I don't use these much in PowerShell but I have used them more in other languages. Consider using an array of arrays when your data fits in a grid like pattern.

Here are two ways we can create a two-dimensional array.

PowerShell

```
$data = @(@(1,2,3),@(4,5,6),@(7,8,9))

$data2 = @(
    @(1,2,3),
    @(4,5,6),
```

```
@(7,8,9)  
)
```

The comma is very important in those examples. I gave an earlier example of a normal array on multiple lines where the comma was optional. That isn't the case with a multi-dimensional array.

The way we use the index notation changes slightly now that we've a nested array. Using the `$data` above, this is how we would access the value 3.

PowerShell

```
PS> $outside = 0  
PS> $inside = 2  
PS> $data[$outside][$inside]  
3
```

Add a set of bracket for each level of array nesting. The first set of brackets is for the outer most array and then you work your way in from there.

Write-Output -NoEnumerate

PowerShell likes to unwrap or enumerate arrays. This is a core aspect of the way PowerShell uses the pipeline but there are times that you don't want that to happen.

I commonly pipe objects to `Get-Member` to learn more about them. When I pipe an array to it, it gets unwrapped and Get-Member sees the members of the array and not the actual array.

PowerShell

```
PS> $data = @('red','green','blue')  
PS> $data | Get-Member  
TypeName: System.String  
...
```

To prevent that unwrap of the array, you can use `Write-Output -NoEnumerate`.

PowerShell

```
PS> Write-Output -NoEnumerate $data | Get-Member  
TypeName: System.Object[]  
...
```

I have a second way that's more of a hack (and I try to avoid hacks like this). You can place a comma in front of the array before you pipe it. This wraps `$data` into another array where it is the only element, so after the unwrapping the outer array we get back `$data` unwrapped.

```
PowerShell
```

```
PS> ,$data | Get-Member
TypeName: System.Object[]
...
...
```

Return an array

This unwrapping of arrays also happens when you output or return values from a function. You can still get an array if you assign the output to a variable so this isn't commonly an issue.

The catch is that you have a new array. If that is ever a problem, you can use `Write-Output -NoEnumerate $array` or `return ,$array` to work around it.

Anything else?

I know this is all a lot to take in. My hope is that you learn something from this article every time you read it and that it turns out to be a good reference for you for a long time to come. If you found this to be helpful, please share it with others you think may get value out of it.

From here, I would recommend you check out a similar post that I wrote about [hashables](#).

Everything you wanted to know about hashtables

Article • 06/26/2023

I want to take a step back and talk about [hashtables](#). I use them all the time now. I was teaching someone about them after our user group meeting last night and I realized I had the same confusion about them as he had. Hashtables are really important in PowerShell so it's good to have a solid understanding of them.

ⓘ Note

The [original version](#) of this article appeared on the blog written by [@KevinMarquette](#). The PowerShell team thanks Kevin for sharing this content with us. Please check out his blog at [PowerShellExplained.com](#).

Hashtable as a collection of things

I want you to first see a **Hashtable** as a collection in the traditional definition of a hashtable. This definition gives you a fundamental understanding of how they work when they get used for more advanced stuff later. Skipping this understanding is often a source of confusion.

What is an array?

Before I jump into what a **Hashtable** is, I need to mention [arrays](#) first. For the purpose of this discussion, an array is a list or collection of values or objects.

```
PowerShell
```

```
$array = @(1,2,3,5,7,11)
```

Once you have your items into an array, you can either use `foreach` to iterate over the list or use an index to access individual elements in the array.

```
PowerShell
```

```
foreach($item in $array)
{
    Write-Output $item
```

```
}
```

```
Write-Output $array[3]
```

You can also update values using an index in the same way.

```
PowerShell
```

```
$array[2] = 13
```

I just scratched the surface on arrays but that should put them into the right context as I move onto hashtables.

What is a hashtable?

I'm going to start with a basic technical description of what hashtables are, in the general sense, before I shift into the other ways PowerShell uses them.

A hashtable is a data structure, much like an array, except you store each value (object) using a key. It's a basic key/value store. First, we create an empty hashtable.

```
PowerShell
```

```
$ageList = @{}  
 
```

Notice that braces, instead of parentheses, are used to define a hashtable. Then we add an item using a key like this:

```
PowerShell
```

```
$key = 'Kevin'  
$value = 36  
$ageList.Add( $key, $value )  
  
$ageList.Add( 'Alex', 9 )
```

The person's name is the key and their age is the value that I want to save.

Using the brackets for access

Once you add your values to the hashtable, you can pull them back out using that same key (instead of using a numeric index like you would have for an array).

```
PowerShell
```

```
$ageList['Kevin']  
$ageList['Alex']
```

When I want Kevin's age, I use his name to access it. We can use this approach to add or update values into the hashtable too. This is just like using the `Add()` method above.

```
PowerShell
```

```
$ageList = @{}  
  
$key = 'Kevin'  
$value = 36  
$ageList[$key] = $value  
  
$ageList['Alex'] = 9
```

There's another syntax you can use for accessing and updating values that I'll cover in a later section. If you're coming to PowerShell from another language, these examples should fit in with how you may have used hashtables before.

Creating hashtables with values

So far I've created an empty hashtable for these examples. You can pre-populate the keys and values when you create them.

```
PowerShell
```

```
$ageList = @{  
    Kevin = 36  
    Alex = 9  
}
```

As a lookup table

The real value of this type of a hashtable is that you can use them as a lookup table. Here is a simple example.

```
PowerShell
```

```
$environments = @{  
    Prod = 'SrvProd05'  
    QA = 'SrvQA02'  
    Dev = 'SrvDev12'
```

```
}
```

```
$server = $environments[$env]
```

In this example, you specify an environment for the `$env` variable and it will pick the correct server. You could use a `switch($env){...}` for a selection like this but a hashtable is a nice option.

This gets even better when you dynamically build the lookup table to use it later. So think about using this approach when you need to cross reference something. I think we would see this even more if PowerShell wasn't so good at filtering on the pipe with `Where-Object`. If you're ever in a situation where performance matters, this approach needs to be considered.

I won't say that it's faster, but it does fit into the rule of [If performance matters, test it ↗](#).

Multiselection

Generally, you think of a hashtable as a key/value pair, where you provide one key and get one value. PowerShell allows you to provide an array of keys to get multiple values.

```
PowerShell
```

```
$environments[@('QA', 'DEV')]
```

```
$environments[('QA', 'DEV')]
```

```
$environments['QA', 'DEV']
```

In this example, I use the same lookup hashtable from above and provide three different array styles to get the matches. This is a hidden gem in PowerShell that most people aren't aware of.

Iterating hashtables

Because a hashtable is a collection of key/value pairs, you iterate over it differently than you do for an array or a normal list of items.

The first thing to notice is that if you pipe your hashtable, the pipe treats it like one object.

```
PowerShell
```

```
PS> $ageList | Measure-Object
```

```
count : 1
```

Even though the `Count` property tells you how many values it contains.

```
PowerShell
```

```
PS> $ageList.Count  
2
```

You get around this issue by using the `Values` property if all you need is just the values.

```
PowerShell
```

```
PS> $ageList.Values | Measure-Object -Average  
Count : 2  
Average : 22.5
```

It's often more useful to enumerate the keys and use them to access the values.

```
PowerShell
```

```
PS> $ageList.Keys | ForEach-Object{  
    $message = '{0} is {1} years old!' -f $_, $ageList[$_]  
    Write-Output $message  
}  
Kevin is 36 years old  
Alex is 9 years old
```

Here is the same example with a `foreach(){...}` loop.

```
PowerShell
```

```
foreach($key in $ageList.Keys)  
{  
    $message = '{0} is {1} years old' -f $key, $ageList[$key]  
    Write-Output $message  
}
```

We are walking each key in the hashtable and then using it to access the value. This is a common pattern when working with hashtables as a collection.

GetEnumerator()

That brings us to `GetEnumerator()` for iterating over our hashtable.

```
PowerShell
```

```
$ageList.GetEnumerator() | ForEach-Object{
    $message = '{0} is {1} years old!' -f $_.Key, $_.Value
    Write-Output $message
}
```

The enumerator gives you each key/value pair one after another. It was designed specifically for this use case. Thank you to [Mark Kraus](#) for reminding me of this one.

BadEnumeration

One important detail is that you can't modify a hashtable while it's being enumerated. If we start with our basic `$environments` example:

PowerShell

```
$environments = @{
    Prod = 'SrvProd05'
    QA   = 'SrvQA02'
    Dev  = 'SrvDev12'
}
```

And trying to set every key to the same server value fails.

PowerShell

```
$environments.Keys | ForEach-Object {
    $environments[$_] = 'SrvDev03'
}
```

```
An error occurred while enumerating through a collection: Collection was modified;
enumeration operation may not execute.
+ CategoryInfo          : InvalidOperation:
tableEnumerator:HashtableEnumerator) [],
RuntimeException
+ FullyQualifiedErrorId : BadEnumeration
```

This will also fail even though it looks like it should also be fine:

PowerShell

```
foreach($key in $environments.Keys) {
    $environments[$key] = 'SrvDev03'
}
```

```
Collection was modified; enumeration operation may not execute.
+ CategoryInfo          : OperationStopped: () [],

```

```
InvalidOperationException  
+ FullyQualifiedErrorId : System.InvalidOperationException
```

The trick to this situation is to clone the keys before doing the enumeration.

PowerShell

```
$environments.Keys.Clone() | ForEach-Object {  
    $environments[$_] = 'SrvDev03'  
}
```

Hashtable as a collection of properties

So far the type of objects we placed in our hashtable were all the same type of object. I used ages in all those examples and the key was the person's name. This is a great way to look at it when your collection of objects each have a name. Another common way to use hashtables in PowerShell is to hold a collection of properties where the key is the name of the property. I'll step into that idea in this next example.

Property-based access

The use of property-based access changes the dynamics of hashtables and how you can use them in PowerShell. Here is our usual example from above treating the keys as properties.

PowerShell

```
$ageList = @{}  
$ageList.Kevin = 35  
$ageList.Alex = 9
```

Just like the examples above, this example adds those keys if they don't exist in the hashtable already. Depending on how you defined your keys and what your values are, this is either a little strange or a perfect fit. The age list example has worked great up until this point. We need a new example for this to feel right going forward.

PowerShell

```
$person = @{  
    name = 'Kevin'  
    age = 36  
}
```

And we can add and access attributes on the `$person` like this.

PowerShell

```
$person.city = 'Austin'  
$person.state = 'TX'
```

All of a sudden this hashtable starts to feel and act like an object. It's still a collection of things, so all the examples above still apply. We just approach it from a different point of view.

Checking for keys and values

In most cases, you can just test for the value with something like this:

PowerShell

```
if( $person.age ){...}
```

It's simple but has been the source of many bugs for me because I was overlooking one important detail in my logic. I started to use it to test if a key was present. When the value was `$false` or zero, that statement would return `$false` unexpectedly.

PowerShell

```
if( $person.age -ne $null ){...}
```

This works around that issue for zero values but not `$null` vs non-existent keys. Most of the time you don't need to make that distinction but there are methods for when you do.

PowerShell

```
if( $person.ContainsKey('age') ){...}
```

We also have a `ContainsValue()` for the situation where you need to test for a value without knowing the key or iterating the whole collection.

Removing and clearing keys

You can remove keys with the `Remove()` method.

```
PowerShell
```

```
$person.Remove('age')
```

Assigning them a `$null` value just leaves you with a key that has a `$null` value.

A common way to clear a hashtable is to just initialize it to an empty hashtable.

```
PowerShell
```

```
$person = @{}
```

While that does work, try to use the `Clear()` method instead.

```
PowerShell
```

```
$person.Clear()
```

This is one of those instances where using the method creates self-documenting code and it makes the intentions of the code very clean.

All the fun stuff

Ordered hashtables

By default, hashtables aren't ordered (or sorted). In the traditional context, the order doesn't matter when you always use a key to access values. You may find that you want the properties to stay in the order that you define them. Thankfully, there's a way to do that with the `ordered` keyword.

```
PowerShell
```

```
$person = [ordered]@{  
    name = 'Kevin'  
    age  = 36  
}
```

Now when you enumerate the keys and values, they stay in that order.

Inline hashtables

When you're defining a hashtable on one line, you can separate the key/value pairs with a semicolon.

```
PowerShell
```

```
$person = @{ name = 'kevin'; age = 36; }
```

This will come in handy if you're creating them on the pipe.

Custom expressions in common pipeline commands

There are a few cmdlets that support the use of hashtables to create custom or calculated properties. You commonly see this with `Select-Object` and `Format-Table`. The hashtables have a special syntax that looks like this when fully expanded.

```
PowerShell
```

```
$property = @{
    Name = 'TotalSpaceGB'
    Expression = { ($_.Used + $_.Free) / 1GB }
}
```

The `Name` is what the cmdlet would label that column. The `Expression` is a script block that is executed where `$_` is the value of the object on the pipe. Here is that script in action:

```
PowerShell
```

```
$drives = Get-PSDrive | where Used
$drives | Select-Object -Property Name, $property

Name      TotalSpaceGB
----      -----
C        238.472652435303
```

I placed that in a variable but it could easily be defined inline and you can shorten `Name` to `n` and `Expression` to `e` while you're at it.

```
PowerShell
```

```
$drives | Select-Object -Property Name, @{n='TotalSpaceGB';e={($_.Used +
$_.Free) / 1GB}}
```

I personally don't like how long that makes commands and it often promotes some bad behaviors that I won't get into. I'm more likely to create a new hashtable or `pscustomobject` with all the fields and properties that I want instead of using this approach in scripts. But there's a lot of code out there that does this so I wanted you to be aware of it. I talk about creating a `pscustomobject` later on.

Custom sort expression

It's easy to sort a collection if the objects have the data that you want to sort on. You can either add the data to the object before you sort it or create a custom expression for `Sort-Object`.

PowerShell

```
Get-ADUser | Sort-Object -Property @{ e={ Get-TotalSales $_.Name } }
```

In this example I'm taking a list of users and using some custom cmdlet to get additional information just for the sort.

Sort a list of Hashtables

If you have a list of hashtables that you want to sort, you'll find that the `Sort-Object` doesn't treat your keys as properties. We can get around that by using a custom sort expression.

PowerShell

```
$data = @(
    @{name='a'}
    @{name='c'}
    @{name='e'}
    @{name='f'}
    @{name='d'}
    @{name='b'}
)
$data | Sort-Object -Property @{e={$_ .name}}
```

Splatting hashtables at cmdlets

This is one of my favorite things about hashtables that many people don't discover early on. The idea is that instead of providing all the properties to a cmdlet on one line, you

can instead pack them into a hashtable first. Then you can give the hashtable to the function in a special way. Here is an example of creating a DHCP scope the normal way.

```
PowerShell
```

```
Add-DhcpServerV4Scope -Name 'TestNetwork' -StartRange '10.0.0.2' -EndRange  
'10.0.0.254' -SubnetMask '255.255.255.0' -Description 'Network for testlab  
A' -LeaseDuration (New-TimeSpan -Days 8) -Type "Both"
```

Without using [splatting](#), all those things need to be defined on a single line. It either scrolls off the screen or will wrap where ever it feels like. Now compare that to a command that uses splatting.

```
PowerShell
```

```
$DHCPScope = @{  
    Name      = 'TestNetwork'  
    StartRange = '10.0.0.2'  
    EndRange   = '10.0.0.254'  
    SubnetMask = '255.255.255.0'  
    Description = 'Network for testlab A'  
    LeaseDuration = (New-TimeSpan -Days 8)  
    Type       = "Both"  
}  
Add-DhcpServerV4Scope @DHCPscope
```

The use of the `@` sign instead of the `$` is what invokes the splat operation.

Just take a moment to appreciate how easy that example is to read. They are the exact same command with all the same values. The second one is easier to understand and maintain going forward.

I use splatting anytime the command gets too long. I define too long as causing my window to scroll right. If I hit three properties for a function, odds are that I'll rewrite it using a splatted hashtable.

Splatting for optional parameters

One of the most common ways I use splatting is to deal with optional parameters that come from some place else in my script. Let's say I have a function that wraps a `Get-CimInstance` call that has an optional `$Credential` argument.

```
PowerShell
```

```
$CIMParams = @{  
    ClassName = 'Win32_BIOS'
```

```

ComputerName = $ComputerName
}

if($Credential)
{
    $CIMParams.Credential = $Credential
}

Get-CimInstance @CIMParams

```

I start by creating my hashtable with common parameters. Then I add the `$Credential` if it exists. Because I'm using splatting here, I only need to have the call to `Get-CimInstance` in my code once. This design pattern is very clean and can handle lots of optional parameters easily.

To be fair, you could write your commands to allow `$null` values for parameters. You just don't always have control over the other commands you're calling.

Multiple splats

You can splat multiple hashtables to the same cmdlet. If we revisit our original splatting example:

```

PowerShell

$Common = @{
    SubnetMask = '255.255.255.0'
    LeaseDuration = (New-TimeSpan -Days 8)
    Type = "Both"
}

$DHCPScope = @{
    Name      = 'TestNetwork'
    StartRange = '10.0.0.2'
    EndRange   = '10.0.0.254'
    Description = 'Network for testlab A'
}

Add-DhcpServerv4Scope @DHCPScope @Common

```

I'll use this method when I have a common set of parameters that I'm passing to lots of commands.

Splatting for clean code

There's nothing wrong with splatting a single parameter if makes you code cleaner.

```
PowerShell
```

```
$log = @{Path = '.\logfile.log'}
Add-Content "logging this command" @log
```

Splatting executables

Splatting also works on some executables that use a `/param:value` syntax. `Robocopy.exe`, for example, has some parameters like this.

```
PowerShell
```

```
$robo = @{R=1;W=1;MT=8}
robocopy source destination @robo
```

I don't know that this is all that useful, but I found it interesting.

Adding hashTables

HashTables support the addition operator to combine two hashTables.

```
PowerShell
```

```
$person += @{Zip = '78701'}
```

This only works if the two hashTables don't share a key.

Nested hashTables

We can use hashTables as values inside a hashtable.

```
PowerShell
```

```
$person = @{
    name = 'Kevin'
    age  = 36
}
$person.location = {}
$person.location.city = 'Austin'
$person.location.state = 'TX'
```

I started with a basic hashtable containing two keys. I added a key called `location` with an empty hashtable. Then I added the last two items to that `location` hashtable. We can do this all inline too.

PowerShell

```
$person = @{
    name = 'Kevin'
    age  = 36
    location = @{
        city  = 'Austin'
        state = 'TX'
    }
}
```

This creates the same hashtable that we saw above and can access the properties the same way.

PowerShell

```
$person.location.city
Austin
```

There are many ways to approach the structure of your objects. Here is a second way to look at a nested hashtable.

PowerShell

```
$people = @{
    Kevin = @{
        age  = 36
        city = 'Austin'
    }
    Alex = @{
        age  = 9
        city = 'Austin'
    }
}
```

This mixes the concept of using hashtables as a collection of objects and a collection of properties. The values are still easy to access even when they're nested using whatever approach you prefer.

PowerShell

```
PS> $people.Kevin.age
36
```

```
PS> $people.kevin['city']
Austin
PS> $people['Alex'].age
9
PS> $people['Alex']['City']
Austin
```

I tend to use the dot property when I'm treating it like a property. Those are generally things I've defined statically in my code and I know them off the top of my head. If I need to walk the list or programmatically access the keys, I use the brackets to provide the key name.

PowerShell

```
foreach($name in $people.Keys)
{
    $person = $people[$name]
    '{0}, age {1}, is in {2}' -f $name, $person.age, $person.city
}
```

Having the ability to nest hashtables gives you a lot of flexibility and options.

Looking at nested hashtables

As soon as you start nesting hashtables, you're going to need an easy way to look at them from the console. If I take that last hashtable, I get an output that looks like this and it only goes so deep:

PowerShell

```
PS> $people
Name          Value
----          -----
Kevin         {age, city}
Alex          {age, city}
```

My go to command for looking at these things is `ConvertTo-Json` because it's very clean and I frequently use JSON on other things.

PowerShell

```
PS> $people | ConvertTo-Json
{
    "Kevin":  {
        "age": 36,
        "city": "Austin"
```

```
        },
    "Alex":  {
        "age":  9,
        "city": "Austin"
    }
}
```

Even if you don't know JSON, you should be able to see what you're looking for. There's a `Format-Custom` command for structured data like this but I still like the JSON view better.

Creating objects

Sometimes you just need to have an object and using a hashtable to hold properties just isn't getting the job done. Most commonly you want to see the keys as column names. A `pscustomobject` makes that easy.

PowerShell

```
$person = [pscustomobject]@{
    name = 'Kevin'
    age  = 36
}

$person

name  age
----  --
Kevin  36
```

Even if you don't create it as a `pscustomobject` initially, you can always cast it later when needed.

PowerShell

```
$person = @{
    name = 'Kevin'
    age  = 36
}

[pscustomobject]$person

name  age
----  --
Kevin  36
```

I already have detailed write-up for [pscustomobject](#) that you should go read after this one. It builds on a lot of the things learned here.

Reading and writing hashtables to file

Saving to CSV

Struggling with getting a hashtable to save to a CSV is one of the difficulties that I was referring to above. Convert your hashtable to a `pscustomobject` and it will save correctly to CSV. It helps if you start with a `pscustomobject` so the column order is preserved. But you can cast it to a `pscustomobject` inline if needed.

PowerShell

```
$person | ForEach-Object{ [pscustomobject]$_. } | Export-Csv -Path $path
```

Again, check out my write-up on using a [pscustomobject](#).

Saving a nested hashtable to file

If I need to save a nested hashtable to a file and then read it back in again, I use the JSON cmdlets to do it.

PowerShell

```
$people | ConvertTo-Json | Set-Content -Path $path  
$people = Get-Content -Path $path -Raw | ConvertFrom-Json
```

There are two important points about this method. First is that the JSON is written out multiline so I need to use the `-Raw` option to read it back into a single string. The Second is that the imported object is no longer a `[hashtable]`. It's now a `[pscustomobject]` and that can cause issues if you don't expect it.

Watch for deeply-nested hashtables. When you convert it to JSON you might not get the results you expect.

PowerShell

```
@{ a = @{ b = @{ c = @{ d = "e" }}} } | ConvertTo-Json  
{  
    "a": {
```

```
        "b": {
            "c": "System.Collections.Hashtable"
        }
    }
}
```

Use **Depth** parameter to ensure that you have expanded all the nested hashtables.

PowerShell

```
@{ a = @{ b = @{ c = @{ d = "e" }}} } | ConvertTo-Json -Depth 3

{
    "a": {
        "b": {
            "c": {
                "d": "e"
            }
        }
    }
}
```

If you need it to be a `[hashtable]` on import, then you need to use the `Export-CliXml` and `Import-CliXml` commands.

Converting JSON to Hashtable

If you need to convert JSON to a `[hashtable]`, there's one way that I know of to do it with the [JavaScriptSerializer](#) in .NET.

PowerShell

```
[Reflection.Assembly]::LoadWithPartialName("System.Web.Script.Serialization")
)
$JSSerializer =
[System.Web.Script.Serialization.JavaScriptSerializer]::new()
$JSSerializer.Deserialize($json, 'Hashtable')
```

Beginning in PowerShell v6, JSON support uses the [NewtonSoft JSON.NET](#) and adds hashtable support.

PowerShell

```
'{ "a": "b" }' | ConvertFrom-Json -AsHashtable
```

Name	Value

```
----      ----  
a          b
```

PowerShell 6.2 added the **Depth** parameter to `ConvertFrom-Json`. The default **Depth** is 1024.

Reading directly from a file

If you have a file that contains a hashtable using PowerShell syntax, there's a way to import it directly.

```
PowerShell
```

```
$content = Get-Content -Path $Path -Raw -ErrorAction Stop  
$scriptBlock = [scriptblock]::Create( $content )  
$scriptBlock.CheckRestrictedLanguage( $allowedCommands, $allowedVariables,  
$true )  
$hashtable = ( & $scriptBlock )
```

It imports the contents of the file into a `scriptblock`, then checks to make sure it doesn't have any other PowerShell commands in it before it executes it.

On that note, did you know that a module manifest (the `.psd1` file) is just a hashtable?

Keys can be any object

Most of the time, the keys are just strings. So we can put quotes around anything and make it a key.

```
PowerShell
```

```
$person = @{  
    'full name' = 'Kevin Marquette'  
    '#' = 3978  
}  
$person['full name']
```

You can do some odd things that you may not have realized you could do.

```
PowerShell
```

```
$person.'full name'
```

```
$key = 'full name'  
$person.$key
```

Just because you can do something, it doesn't mean that you should. That last one just looks like a bug waiting to happen and would be easily misunderstood by anyone reading your code.

Technically your key doesn't have to be a string but they're easier to think about if you only use strings. However, indexing doesn't work well with the complex keys.

PowerShell

```
$ht = @{ @(1,2,3) = "a" }  
$ht  
  
Name          Value  
----          ----  
{1, 2, 3}      a
```

Accessing a value in the hashtable by its key doesn't always work. For example:

PowerShell

```
$key = $ht.Keys[0]  
$ht.$($key)  
a  
$ht[$key]  
a
```

When the key is an array, you must wrap the `$key` variable in a subexpression so that it can be used with member access (`.`) notation. Or, you can use array index (`[]`) notation.

Use in automatic variables

\$PSBoundParameters

[\\$PSBoundParameters](#) is an automatic variable that only exists inside the context of a function. It contains all the parameters that the function was called with. This isn't exactly a hashtable but close enough that you can treat it like one.

That includes removing keys and splatting it to other functions. If you find yourself writing proxy functions, take a closer look at this one.

See [about_Automatic_Variables](#) for more details.

PSBoundParameters gotcha

One important thing to remember is that this only includes the values that are passed in as parameters. If you also have parameters with default values but aren't passed in by the caller, `$PSBoundParameters` doesn't contain those values. This is commonly overlooked.

\$PSDefaultParameterValues

This automatic variable lets you assign default values to any cmdlet without changing the cmdlet. Take a look at this example.

PowerShell

```
$PSDefaultParameterValues["Out-File:Encoding"] = "UTF8"
```

This adds an entry to the `$PSDefaultParameterValues` hashtable that sets `UTF8` as the default value for the `Out-File -Encoding` parameter. This is session-specific so you should place it in your `$PROFILE`.

I use this often to pre-assign values that I type quite often.

PowerShell

```
$PSDefaultParameterValues[ "Connect-VIServer:Server" ] =  
'VCENTER01.contoso.local'
```

This also accepts wildcards so you can set values in bulk. Here are some ways you can use that:

PowerShell

```
$PSDefaultParameterValues[ "Get-*:Verbose" ] = $true  
$PSDefaultParameterValues[ "*:Credential" ] = Get-Credential
```

For a more in-depth breakdown, see this great article on [Automatic Defaults ↗](#) by Michael Sorens ↗.

Regex \$Matches

When you use the `-match` operator, an automatic variable called `$Matches` is created with the results of the match. If you have any sub expressions in your regex, those sub matches are also listed.

PowerShell

```
$message = 'My SSN is 123-45-6789.'  
  
$message -match 'My SSN is (.)\.'  
$Matches[0]  
$Matches[1]
```

Named matches

This is one of my favorite features that most people don't know about. If you use a named regex match, then you can access that match by name on the matches.

PowerShell

```
$message = 'My Name is Kevin and my SSN is 123-45-6789.'  
  
if($message -match 'My Name is (?<Name>.+) and my SSN is (?<SSN>.+)\.')  
{  
    $Matches.Name  
    $Matches.SSN  
}
```

In the example above, the `(?<Name>.*)` is a named sub expression. This value is then placed in the `$Matches.Name` property.

Group-Object -AsHashtable

One little known feature of `Group-Object` is that it can turn some datasets into a hashtable for you.

PowerShell

```
Import-Csv $Path | Group-Object -AsHashtable -Property Email
```

This will add each row into a hashtable and use the specified property as the key to access it.

Copying Hashtables

One important thing to know is that hashtables are objects. And each variable is just a reference to an object. This means that it takes more work to make a valid copy of a hashtable.

Assigning reference types

When you have one hashtable and assign it to a second variable, both variables point to the same hashtable.

PowerShell

```
PS> $orig = @{name='orig'}
PS> $copy = $orig
PS> $copy.name = 'copy'
PS> 'Copy: [{0}]' -f $copy.name
PS> 'Orig: [{0}]' -f $orig.name

Copy: [copy]
Orig: [copy]
```

This highlights that they're the same because altering the values in one will also alter the values in the other. This also applies when passing hashtables into other functions. If those functions make changes to that hashtable, your original is also altered.

Shallow copies, single level

If we have a simple hashtable like our example above, we can use `Clone()` to make a shallow copy.

PowerShell

```
PS> $orig = @{name='orig'}
PS> $copy = $orig.Clone()
PS> $copy.name = 'copy'
PS> 'Copy: [{0}]' -f $copy.name
PS> 'Orig: [{0}]' -f $orig.name

Copy: [copy]
Orig: [orig]
```

This will allow us to make some basic changes to one that don't impact the other.

Shallow copies, nested

The reason why it's called a shallow copy is because it only copies the base level properties. If one of those properties is a reference type (like another hashtable), then those nested objects will still point to each other.

```
PowerShell

PS> $orig = @{
    person=@{
        name='orig'
    }
}
PS> $copy = $orig.Clone()
PS> $copy.person.name = 'copy'
PS> 'Copy: [{0}]' -f $copy.person.name
PS> 'Orig: [{0}]' -f $orig.person.name

Copy: [copy]
Orig: [copy]
```

So you can see that even though I cloned the hashtable, the reference to `person` wasn't cloned. We need to make a deep copy to truly have a second hashtable that isn't linked to the first.

Deep copies

There are a couple of ways to make a deep copy of a hashtable (and keep it as a hashtable). Here's a function using PowerShell to recursively create a deep copy:

```
PowerShell

function Get-DeepClone
{
    [CmdletBinding()]
    param(
        $InputObject
    )
    process
    {
        if($InputObject -is [hashtable]) {
            $clone = @{}
            foreach($key in $InputObject.Keys)
            {
                $clone[$key] = Get-DeepClone $InputObject[$key]
            }
            return $clone
        } else {
            return $InputObject
        }
    }
}
```

```
        }
    }
}
```

It doesn't handle any other reference types or arrays, but it's a good starting point.

Another way is to use .NET to deserialize it using **CliXml** like in this function:

PowerShell

```
function Get-DeepClone
{
    param(
        $InputObject
    )
    $TempCliXmlString =
[System.Management.Automation.PSSerializer]::Serialize($obj,
[int32]::.MaxValue)
    return
[System.Management.Automation.PSSerializer]::Deserialize($TempCliXmlString)
}
```

For extremely large hashtables, the deserializing function is faster as it scales out. However, there are some things to consider when using this method. Since it uses **CliXml**, it's memory intensive and if you are cloning huge hashtables, that might be a problem. Another limitation of the **CliXml** is there is a depth limitation of 48. Meaning, if you have a hashtable with 48 layers of nested hashtables, the cloning will fail and no hashtable will be output at all.

Anything else?

I covered a lot of ground quickly. My hope is that you walk away leaning something new or understanding it better every time you read this. Because I covered the full spectrum of this feature, there are aspects that just may not apply to you right now. That is perfectly OK and is kind of expected depending on how much you work with PowerShell.

Everything you wanted to know about PSCustomObject

Article • 06/20/2024

`PSCustomObject` is a great tool to add into your PowerShell tool belt. Let's start with the basics and work our way into the more advanced features. The idea behind using a `PSCustomObject` is to have a simple way to create structured data. Take a look at the first example and you'll have a better idea of what that means.

ⓘ Note

The [original version](#) of this article appeared on the blog written by [@KevinMarquette](#). The PowerShell team thanks Kevin for sharing this content with us. Please check out his blog at [PowerShellExplained.com](#).

Creating a PSCustomObject

I love using `[pscustomobject]` in PowerShell. Creating a usable object has never been easier. Because of that, I'm going to skip over all the other ways you can create an object but I need to mention that most of these examples are PowerShell v3.0 and newer.

PowerShell

```
$myObject = [pscustomobject]@{  
    Name      = 'Kevin'  
    Language  = 'PowerShell'  
    State     = 'Texas'  
}
```

This method works well for me because I use hashtables for just about everything. But there are times when I would like PowerShell to treat hashtables more like an object. The first place you notice the difference is when you want to use `Format-Table` or `Export-Csv` and you realize that a hashtable is just a collection of key/value pairs.

You can then access and use the values like you would a normal object.

PowerShell

```
$myObject.Name
```

Converting a hashtable

While I am on the topic, did you know you could do this:

PowerShell

```
$myHashtable = @{
    Name      = 'Kevin'
    Language = 'PowerShell'
    State     = 'Texas'
}
$myObject = [pscustomobject]$myHashtable
```

I do prefer to create the object from the start but there are times you have to work with a hashtable first. This example works because the constructor takes a hashtable for the object properties. One important note is that while this method works, it isn't an exact equivalent. The biggest difference is that the order of the properties isn't preserved.

If you want to preserve the order, see [Ordered hashtables](#).

Legacy approach

You may have seen people use `New-Object` to create custom objects.

PowerShell

```
$myHashtable = @{
    Name      = 'Kevin'
    Language = 'PowerShell'
    State     = 'Texas'
}

$myObject = New-Object -TypeName psobject -Property $myHashtable
```

This way is quite a bit slower but it may be your best option on early versions of PowerShell.

Saving to a file

I find the best way to save a hashtable to a file is to save it as JSON. You can import it back into a `[pscustomobject]`

```
PowerShell
```

```
$myObject | ConvertTo-Json -Depth 1 | Set-Content -Path $Path  
$myObject = Get-Content -Path $Path | ConvertFrom-Json
```

I cover more ways to save objects to a file in my article on [The many ways to read and write to files ↗](#).

Working with properties

Adding properties

You can still add new properties to your `PSCustomObject` with `Add-Member`.

```
PowerShell
```

```
$myObject | Add-Member -MemberType NoteProperty -Name 'ID' -Value  
'KevinMarquette'  
  
$myObject.ID
```

Remove properties

You can also remove properties off of an object.

```
PowerShell
```

```
$myObject.psobject.Properties.Remove('ID')
```

The `.psobject` is an intrinsic member that gives you access to base object metadata. For more information about intrinsic members, see [about_Intrinsic_Members](#).

Enumerating property names

Sometimes you need a list of all the property names on an object.

```
PowerShell
```

```
$myObject | Get-Member -MemberType NoteProperty | select -ExpandProperty  
Name
```

We can get this same list off of the `psobject` property too.

```
PowerShell
```

```
$myobject.psobject.Properties.Name
```

 **Note**

`Get-Member` returns the properties in alphabetical order. Using the member-access operator to enumerate the property names returns the properties in the order they were defined on the object.

Dynamically accessing properties

I already mentioned that you can access property values directly.

```
PowerShell
```

```
$myObject.Name
```

You can use a string for the property name and it will still work.

```
PowerShell
```

```
$myObject.'Name'
```

We can take this one more step and use a variable for the property name.

```
PowerShell
```

```
$property = 'Name'  
$myObject.$property
```

I know that looks strange, but it works.

Convert `PSCustomObject` into a hashtable

To continue on from the last section, you can dynamically walk the properties and create a hashtable from them.

```
PowerShell
```

```
$hashtable = @{}
foreach( $property in $myobject.psobject.Properties.Name )
{
    $hashtable[$property] = $myObject.$property
}
```

Testing for properties

If you need to know if a property exists, you could just check for that property to have a value.

PowerShell

```
if( $null -ne $myObject.ID )
```

But if the value could be `$null` you can check to see if it exists by checking the `psobject.Properties` for it.

PowerShell

```
if( $myobject.psobject.Properties.Match('ID').Count )
```

Adding object methods

If you need to add a script method to an object, you can do it with `Add-Member` and a `ScriptBlock`. You have to use the `this` automatic variable reference the current object. Here is a `scriptblock` to turn an object into a hashtable. (same code form the last example)

PowerShell

```
$ScriptBlock = {
    $hashtable = @{}
    foreach( $property in $this.psobject.Properties.Name )
    {
        $hashtable[$property] = $this.$property
    }
    return $hashtable
}
```

Then we add it to our object as a script property.

```
PowerShell
```

```
$memberParam = @{
    MemberType = "ScriptMethod"
    InputObject = $myobject
    Name = "ToHashtable"
    Value = $scriptBlock
}
Add-Member @memberParam
```

Then we can call our function like this:

```
PowerShell
```

```
$myObject.ToHashtable()
```

Objects vs Value types

Objects and value types don't handle variable assignments the same way. If you assign value types to each other, only the value get copied to the new variable.

```
PowerShell
```

```
$first = 1
$second = $first
$second = 2
```

In this case, `$first` is 1 and `$second` is 2.

Object variables hold a reference to the actual object. When you assign one object to a new variable, they still reference the same object.

```
PowerShell
```

```
$third = [pscustomobject]@{Key=3}
$fourth = $third
$fourth.Key = 4
```

Because `$third` and `$fourth` reference the same instance of an object, both `$third.Key` and `$fourth.Key` are 4.

psobject.Copy()

If you need a true copy of an object, you can clone it.

PowerShell

```
$third = [pscustomobject]@{Key=3}
$fourth = $third.psobject.Copy()
$fourth.Key = 4
```

Clone creates a shallow copy of the object. They have different instances now and `$third.Key` is 3 and `$fourth.Key` is 4 in this example.

I call this a shallow copy because if you have nested objects (objects with properties contain other objects), only the top-level values are copied. The child objects will reference each other.

PSTypeName for custom object types

Now that we have an object, there are a few more things we can do with it that may not be nearly as obvious. First thing we need to do is give it a `PSTypeName`. This is the most common way I see people do it:

PowerShell

```
$myObject.psobject.TypeNames.Insert(0, "My.Object")
```

I recently discovered another way to do this from Redditor [u/markekraus](#). He talks about this approach that allows you to define it inline.

PowerShell

```
$myObject = [pscustomobject]@{
    PSTypeName = 'My.Object'
    Name      = 'Kevin'
    Language   = 'PowerShell'
    State     = 'Texas'
}
```

I love how nicely this just fits into the language. Now that we have an object with a proper type name, we can do some more things.

ⓘ Note

You can also create custom PowerShell types using PowerShell classes. For more information, see [PowerShell Class Overview](#).

Using DefaultPropertySet (the long way)

PowerShell decides for us what properties to display by default. A lot of the native commands have a `.ps1xml` [formatting file](#) that does all the heavy lifting. From this [post by Boe Prox](#), there's another way for us to do this on our custom object using just PowerShell. We can give it a `MemberSet` for it to use.

PowerShell

```
$defaultDisplaySet = 'Name', 'Language'  
$defaultDisplayPropertySet = New-Object  
System.Management.Automation.PSPropertySet('DefaultDisplayPropertySet',  
[string[]]$defaultDisplaySet)  
$PSStandardMembers =  
[System.Management.Automation.PSMemberInfo[]]@($defaultDisplayPropertySet)  
$MyObject | Add-Member MemberSet PSStandardMembers $PSStandardMembers
```

Now when my object just falls to the shell, it will only show those properties by default.

Update-TypeData with DefaultPropertySet

This is nice but I recently saw a better way using `Update-TypeData` to specify the default properties.

PowerShell

```
$TypeData = @{  
    TypeName = 'My.Object'  
    DefaultDisplayPropertySet = 'Name', 'Language'  
}  
Update-TypeData @TypeData
```

That is simple enough that I could almost remember it if I didn't have this post as a quick reference. Now I can easily create objects with lots of properties and still give it a nice clean view when looking at it from the shell. If I need to access or see those other properties, they're still there.

PowerShell

```
$myObject | Format-List *
```

Update-TypeData with ScriptProperty

Something else I got out of that video was creating script properties for your objects. This would be a good time to point out that this works for existing objects too.

```
PowerShell

$TypeData = @{
    TypeName = 'My.Object'
    MemberType = 'ScriptProperty'
    MemberName = 'UpperCaseName'
    Value = {$this.Name.ToUpper()}
}
Update-TypeData @TypeData
```

You can do this before your object is created or after and it will still work. This is what makes this different than using `Add-Member` with a script property. When you use `Add-Member` the way I referenced earlier, it only exists on that specific instance of the object. This one applies to all objects with this `TypeName`.

Function parameters

You can now use these custom types for parameters in your functions and scripts. You can have one function create these custom objects and then pass them into other functions.

```
PowerShell

param( [PSTypeName('My.Object')]$Data )
```

PowerShell requires that the object is the type you specified. It throws a validation error if the type doesn't match automatically to save you the step of testing for it in your code. A great example of letting PowerShell do what it does best.

Function OutputType

You can also define an `OutputType` for your advanced functions.

```
PowerShell

function Get-MyObject
{
    [OutputType('My.Object')]
    [CmdletBinding()]
    param
```

```
(  
...)
```

The **OutputType** attribute value is only a documentation note. It isn't derived from the function code or compared to the actual function output.

The main reason you would use an output type is so that meta information about your function reflects your intentions. Things like `Get-Command` and `Get-Help` that your development environment can take advantage of. If you want more information, then take a look at the help for it: [about_Functions_OutputTypeAttribute](#).

With that said, if you're using Pester to unit test your functions then it would be a good idea to validate the output objects match your **OutputType**. This could catch variables that just fall to the pipe when they shouldn't.

Closing thoughts

The context of this was all about `[pscustomobject]`, but a lot of this information applies to objects in general.

I have seen most of these features in passing before but never saw them presented as a collection of information on `PSCustomObject`. Just this last week I stumbled upon another one and was surprised that I had not seen it before. I wanted to pull all these ideas together so you can hopefully see the bigger picture and be aware of them when you have an opportunity to use them. I hope you learned something and can find a way to work this into your scripts.

Everything you wanted to know about variable substitution in strings

Article • 06/20/2024

There are many ways to use variables in strings. I'm calling this variable substitution but I'm referring to any time you want to format a string to include values from variables. This is something that I often find myself explaining to new scripters.

ⓘ Note

The [original version](#) of this article appeared on the blog written by [@KevinMarquette](#). The PowerShell team thanks Kevin for sharing this content with us. Please check out his blog at [PowerShellExplained.com](#).

Concatenation

The first class of methods can be referred to as concatenation. It's basically taking several strings and joining them together. There's a long history of using concatenation to build formatted strings.

PowerShell

```
$name = 'Kevin Marquette'  
$message = 'Hello, ' + $name
```

Concatenation works out OK when there are only a few values to add. But this can get complicated quickly.

PowerShell

```
$first = 'Kevin'  
$last = 'Marquette'
```

PowerShell

```
$message = 'Hello, ' + $first + ' ' + $last + '.'
```

This simple example is already getting harder to read.

Variable substitution

PowerShell has another option that is easier. You can specify your variables directly in the strings.

```
PowerShell
```

```
$message = "Hello, $first $last."
```

The type of quotes you use around the string makes a difference. A double quoted string allows the substitution but a single quoted string doesn't. There are times you want one or the other so you have an option.

Command substitution

Things get a little tricky when you start trying to get the values of properties into a string. This is where many new people get tripped up. First let me show you what they think should work (and at face value almost looks like it should).

```
PowerShell
```

```
$directory = Get-Item 'C:\windows'  
$message = "Time: $directory.CreationTime"
```

You would be expecting to get the `CreationTime` off of the `$directory`, but instead you get this `Time: C:\windows.CreationTime` as your value. The reason is that this type of substitution only sees the base variable. It considers the period as part of the string so it stops resolving the value any deeper.

It just so happens that this object gives a string as a default value when placed into a string. Some objects give you the type name instead like `System.Collections.Hashtable`. Just something to watch for.

PowerShell allows you to do command execution inside the string with a special syntax. This allows us to get the properties of these objects and run any other command to get a value.

```
PowerShell
```

```
$message = "Time: $($directory.CreationTime)"
```

This works great for some situations but it can get just as crazy as concatenation if you have just a few variables.

Command execution

You can run commands inside a string. Even though I have this option, I don't like it. It gets cluttered quickly and hard to debug. I either run the command and save to a variable or use a format string.

```
PowerShell
```

```
$message = "Date: $(Get-Date)"
```

Format string

.NET has a way to format strings that I find fairly easy to work with. First let me show you the static method for it before I show you the PowerShell shortcut to do the same thing.

```
PowerShell
```

```
# .NET string format string
[string]::Format('Hello, {0} {1}.',$first,$last)

# PowerShell format string
'Hello, {0} {1}.' -f $first, $last
```

What is happening here is that the string is parsed for the tokens `{0}` and `{1}`, then it uses that number to pick from the values provided. If you want to repeat one value some place in the string, you can reuse that values number.

The more complicated the string gets, the more value you get out of this approach.

Format values as arrays

If your format line gets too long, you can place your values into an array first.

```
PowerShell
```

```
$values = @(
    "Kevin"
    "Marquette"
)
'Hello, {0} {1}.' -f $values
```

This is not splatting because I'm passing the whole array in, but the idea is similar.

Advanced formatting

I intentionally called these out as coming from .NET because there are lots of formatting options already well [documented](#) on it. There are built-in ways to format various data types.

PowerShell

```
"{0:yyyyMMdd}" -f (Get-Date)  
"Population {0:N0}" -f 8175133
```

Output

```
20211110  
Population 8,175,133
```

I'm not going to go into them but I just wanted to let you know that this is a very powerful formatting engine if you need it.

Joining strings

Sometimes you actually do want to concatenate a list of values together. There's a `-join` operator that can do that for you. It even lets you specify a character to join between the strings.

PowerShell

```
$servers = @(  
    'server1'  
    'server2'  
    'server3'  
)  
  
$servers -join ','
```

If you want to `-join` some strings without a separator, you need to specify an empty string `''`. But if that is all you need, there's a faster option.

PowerShell

```
[string]::Concat('server1','server2','server3')
[string]::Concat($servers)
```

It's also worth pointing out that you can also `-split` strings too.

Join-Path

This is often overlooked but a great cmdlet for building a file path.

PowerShell

```
$folder = 'Temp'
Join-Path -Path 'C:\windows' -ChildPath $folder
```

The great thing about this is it works out the backslashes correctly when it puts the values together. This is especially important if you are taking values from users or config files.

This also goes well with `Split-Path` and `Test-Path`. I also cover these in my post about [reading and saving to files ↗](#).

Strings are arrays

I do need to mention adding strings here before I go on. Remember that a string is just an array of characters. When you add multiple strings together, a new array is created each time.

Look at this example:

PowerShell

```
$message = "Numbers: "
foreach($number in 1..10000)
{
    $message += " $number"
}
```

It looks very basic but what you don't see is that each time a string is added to `$message` that a whole new string is created. Memory gets allocated, data gets copied and the old one is discarded. Not a big deal when it's only done a few times, but a loop like this would really expose the issue.

StringBuilder

StringBuilder is also very popular for building large strings from lots of smaller strings. The reason why is because it just collects all the strings you add to it and only concatenates all of them at the end when you retrieve the value.

PowerShell

```
$stringBuilder = New-Object -TypeName "System.Text.StringBuilder"

[void]$stringBuilder.Append("Numbers: ")
foreach($number in 1..10000)
{
    [void]$stringBuilder.Append(" $number")
}
$message = $stringBuilder.ToString()
```

Again, this is something that I'm reaching out to .NET for. I don't use it often anymore but it's good to know it's there.

Delineation with braces

This is used for suffix concatenation within the string. Sometimes your variable doesn't have a clean word boundary.

PowerShell

```
$test = "Bet"
$tester = "Better"
Write-Host "$test $tester ${test}ter"
```

Thank you Redditor [u/real_parbold](#) for that one.

Here is an alternate to this approach:

PowerShell

```
Write-Host "$test $tester $($test)ter"
Write-Host "{0} {1} {0}ter" -f $test, $tester
```

I personally use format string for this, but this is good to know incase you see it in the wild.

Find and replace tokens

While most of these features limit your need to roll your own solution, there are times where you may have large template files where you want to replace strings inside.

Let us assume you pulled in a template from a file that has a lot of text.

```
PowerShell
```

```
$letter = Get-Content -Path TemplateLetter.txt -RAW  
$letter = $letter -replace '#FULL_NAME#', 'Kevin Marquette'
```

You may have lots of tokens to replace. The trick is to use a very distinct token that is easy to find and replace. I tend to use a special character at both ends to help distinguish it.

I recently found a new way to approach this. I decided to leave this section in here because this is a pattern that is commonly used.

Replace multiple tokens

When I have a list of tokens that I need to replace, I take a more generic approach. I would place them in a hashtable and iterate over them to do the replace.

```
PowerShell
```

```
$tokenList = @{  
    Full_Name = 'Kevin Marquette'  
    Location = 'Orange County'  
    State = 'CA'  
}  
  
$letter = Get-Content -Path TemplateLetter.txt -RAW  
foreach( $token in $tokenList.GetEnumerator() )  
{  
    $pattern = '#{0}#' -f $token.key  
    $letter = $letter -replace $pattern, $token.Value  
}
```

Those tokens could be loaded from JSON or CSV if needed.

ExecutionContext ExpandString

There's a clever way to define a substitution string with single quotes and expand the variables later. Look at this example:

```
PowerShell
```

```
$message = 'Hello, $Name!'
$name = 'Kevin Marquette'
$string = $ExecutionContext.InvokeCommand.ExpandString($message)
```

The call to `InvokeCommand.ExpandString` on the current execution context uses the variables in the current scope for substitution. The key thing here is that the `$message` can be defined very early before the variables even exist.

If we expand on that just a little bit, we can perform this substitution over and over with different values.

PowerShell

```
$message = 'Hello, $Name!'
$nameList = 'Mark Kraus','Kevin Marquette','Lee Dailey'
foreach($name in $nameList){
    $ExecutionContext.InvokeCommand.ExpandString($message)
}
```

To keep going on this idea; you could be importing a large email template from a text file to do this. I have to thank [Mark Kraus ↗](#) for this suggestion.

Whatever works the best for you

I'm a fan of the format string approach. I definitely do this with the more complicated strings or if there are multiple variables. On anything that is very short, I may use any one of these.

Anything else?

I covered a lot of ground on this one. My hope is that you walk away learning something new.

Links

If you'd like to learn more about the methods and features that make string interpolation possible, see the following list for the reference documentation.

- Concatenation uses the [addition operator](#)
- Variable and command substitution follow the [quoting rules](#)
- Formatting uses the [format operator](#)

- Joining strings uses the [join operator](#) and references [Join-Path](#), but you could also read about [Join-String](#)
- Arrays are documented in [About arrays](#)
- `StringBuilder` is a .NET class, with its [own documentation](#)
- Braces in strings is also covered in the [quoting rules](#)
- Token replacement uses the [replace operator](#)
- The `$ExecutionContext.InvokeCommand.ExpandString()` method has [.NET API reference documentation](#)

Everything you wanted to know about the `if` statement

Article • 12/18/2023

Like many other languages, PowerShell has statements for conditionally executing code in your scripts. One of those statements is the `If` statement. Today we will take a deep dive into one of the most fundamental commands in PowerShell.

ⓘ Note

The [original version](#) of this article appeared on the blog written by [@KevinMarquette](#). The PowerShell team thanks Kevin for sharing this content with us. Please check out his blog at [PowerShellExplained.com](#).

Conditional execution

Your scripts often need to make decisions and perform different logic based on those decisions. This is what I mean by conditional execution. You have one statement or value to evaluate, then execute a different section of code based on that evaluation. This is exactly what the `if` statement does.

The `if` statement

Here is a basic example of the `if` statement:

```
PowerShell

$condition = $true
if ( $condition )
{
    Write-Output "The condition was true"
}
```

The first thing the `if` statement does is evaluate the expression in parentheses. If it evaluates to `$true`, then it executes the `scriptblock` in the braces. If the value was `$false`, then it would skip over that scriptblock.

In the previous example, the `if` statement was just evaluating the `$condition` variable. It was `$true` and would have executed the `Write-Output` command inside the

scriptblock.

In some languages, you can place a single line of code after the `if` statement and it gets executed. That isn't the case in PowerShell. You must provide a full `scriptblock` with braces for it to work correctly.

Comparison operators

The most common use of the `if` statement for is comparing two items with each other. PowerShell has special operators for different comparison scenarios. When you use a comparison operator, the value on the left-hand side is compared to the value on the right-hand side.

-eq for equality

The `-eq` does an equality check between two values to make sure they're equal to each other.

PowerShell

```
$value = Get-MysteryValue
if ( 5 -eq $value )
{
    # do something
}
```

In this example, I'm taking a known value of `5` and comparing it to my `$value` to see if they match.

One possible use case is to check the status of a value before you take an action on it. You could get a service and check that the status was running before you called `Restart-Service` on it.

It's common in other languages like C# to use `==` for equality (ex: `5 == $value`) but that doesn't work with PowerShell. Another common mistake that people make is to use the equals sign (ex: `5 = $value`) that is reserved for assigning values to variables. By placing your known value on the left, it makes that mistake more awkward to make.

This operator (and others) has a few variations.

- `-eq` case-insensitive equality
- `-ieq` case-insensitive equality
- `-ceq` case-sensitive equality

-ne not equal

Many operators have a related operator that is checking for the opposite result. `-ne` verifies that the values don't equal each other.

PowerShell

```
if ( 5 -ne $value )  
{  
    # do something  
}
```

Use this to make sure that the action only executes if the value isn't `5`. A good use-cases where would be to check if a service was in the running state before you try to start it.

Variations:

- `-ne` case-insensitive not equal
- `-ine` case-insensitive not equal
- `-cne` case-sensitive not equal

These are inverse variations of `-eq`. I'll group these types together when I list variations for other operators.

-gt -ge -lt -le for greater than or less than

These operators are used when checking to see if a value is larger or smaller than another value. The `-gt -ge -lt -le` stand for GreaterThan, GreaterThanOrEqual, LessThan, and LessThanOrEqual.

PowerShell

```
if ( $value -gt 5 )  
{  
    # do something  
}
```

Variations:

- `-gt` greater than
- `-igt` greater than, case-insensitive
- `-cgt` greater than, case-sensitive
- `-ge` greater than or equal

- `-ige` greater than or equal, case-insensitive
- `-cge` greater than or equal, case-sensitive
- `-lt` less than
- `-ilt` less than, case-insensitive
- `-clt` less than, case-sensitive
- `-le` less than or equal
- `-ile` less than or equal, case-insensitive
- `-cle` less than or equal, case-sensitive

I don't know why you would use case-sensitive and insensitive options for these operators.

-like wildcard matches

PowerShell has its own wildcard-based pattern matching syntax and you can use it with the `-like` operator. These wildcard patterns are fairly basic.

- `?` matches any single character
- `*` matches any number of characters

PowerShell

```
$value = 'S-ATX-SQL01'
if ( $value -like 'S-*SQL??' )
{
    # do something
}
```

It's important to point out that the pattern matches the whole string. If you need to match something in the middle of the string, you need to place the `*` on both ends of the string.

PowerShell

```
$value = 'S-ATX-SQL02'
if ( $value -like '*SQL*' )
{
    # do something
}
```

Variations:

- `-like` case-insensitive wildcard

- `-ilike` case-insensitive wildcard
- `-clike` case-sensitive wildcard
- `-notlike` case-insensitive wildcard not matched
- `-inotlike` case-insensitive wildcard not matched
- `-cnotlike` case-sensitive wildcard not matched

-match regular expression

The `-match` operator allows you to check a string for a regular-expression-based match. Use this when the wildcard patterns aren't flexible enough for you.

PowerShell

```
$value = 'S-ATX-SQL01'
if ( $value -match 'S-\w\w\w-SQL\d\d' )
{
    # do something
}
```

A regex pattern matches anywhere in the string by default. So you can specify a substring that you want matched like this:

PowerShell

```
$value = 'S-ATX-SQL01'
if ( $value -match 'SQL' )
{
    # do something
}
```

Regex is a complex language of its own and worth looking into. I talk more about `-match` and [the many ways to use regex ↗](#) in another article.

Variations:

- `-match` case-insensitive regex
- `-imatch` case-insensitive regex
- `-cmatch` case-sensitive regex
- `-notmatch` case-insensitive regex not matched
- `-inotmatch` case-insensitive regex not matched
- `-cnotmatch` case-sensitive regex not matched

-is of type

You can check a value's type with the `-is` operator.

PowerShell

```
if ( $value -is [string] )
{
    # do something
}
```

You may use this if you're working with classes or accepting various objects over the pipeline. You could have either a service or a service name as your input. Then check to see if you have a service and fetch the service if you only have the name.

PowerShell

```
if ( $Service -isnot [System.ServiceProcess.ServiceController] )
{
    $Service = Get-Service -Name $Service
}
```

Variations:

- `-is` of type
- `-isnot` not of type

Collection operators

When you use the previous operators with a single value, the result is `$true` or `$false`. This is handled slightly differently when working with a collection. Each item in the collection gets evaluated and the operator returns every value that evaluates to `$true`.

PowerShell

```
PS> 1,2,3,4 -eq 3
3
```

This still works correctly in an `if` statement. So a value is returned by your operator, then the whole statement is `$true`.

PowerShell

```
$array = 1..6
if ( $array -gt 3 )
{
```

```
# do something  
}
```

There's one small trap hiding in the details here that I need to point out. When using the `-ne` operator this way, it's easy to mistakenly look at the logic backwards. Using `-ne` with a collection returns `$true` if any item in the collection doesn't match your value.

PowerShell

```
PS> 1,2,3 -ne 4  
1  
2  
3
```

This may look like a clever trick, but we have operators `-contains` and `-in` that handle this more efficiently. And `-notcontains` does what you expect.

-contains

The `-contains` operator checks the collection for your value. As soon as it finds a match, it returns `$true`.

PowerShell

```
$array = 1..6  
if ( $array -contains 3 )  
{  
    # do something  
}
```

This is the preferred way to see if a collection contains your value. Using `Where-Object` (or `-eq`) walks the entire list every time and is significantly slower.

Variations:

- `-contains` case-insensitive match
- `-icontains` case-insensitive match
- `-cccontains` case-sensitive match
- `-notcontains` case-insensitive not matched
- `-inotcontains` case-insensitive not matched
- `-cnotcontains` case-sensitive not matched

-in

The `-in` operator is just like the `-contains` operator except the collection is on the right-hand side.

PowerShell

```
$array = 1..6
if ( 3 -in $array )
{
    # do something
}
```

Variations:

- `-in` case-insensitive match
- `-iin` case-insensitive match
- `-cin` case-sensitive match
- `-notin` case-insensitive not matched
- `-inotin` case-insensitive not matched
- `-cnotin` case-sensitive not matched

Logical operators

Logical operators are used to invert or combine other expressions.

-not

The `-not` operator flips an expression from `$false` to `$true` or from `$true` to `$false`. Here is an example where we want to perform an action when `Test-Path` is `$false`.

PowerShell

```
if ( -not ( Test-Path -Path $path ) )
```

Most of the operators we talked about do have a variation where you do not need to use the `-not` operator. But there are still times it is useful.

! operator

You can use `!` as an alias for `-not`.

PowerShell

```
if ( -not $value ){}  
if ( !$value ){}
```

You may see `!` used more by people that come from another languages like C#. I prefer to type it out because I find it hard to see when quickly looking at my scripts.

-and

You can combine expressions with the `-and` operator. When you do that, both sides need to be `$true` for the whole expression to be `$true`.

PowerShell

```
if ( ($age -gt 13) -and ($age -lt 55) )
```

In that example, `$age` must be 13 or older for the left side and less than 55 for the right side. I added extra parentheses to make it clearer in that example but they're optional as long as the expression is simple. Here is the same example without them.

PowerShell

```
if ( $age -gt 13 -and $age -lt 55 )
```

Evaluation happens from left to right. If the first item evaluates to `$false`, it exits early and doesn't perform the right comparison. This is handy when you need to make sure a value exists before you use it. For example, `Test-Path` throws an error if you give it a `$null` path.

PowerShell

```
if ( $null -ne $path -and (Test-Path -Path $path) )
```

-Or

The `-or` allows for you to specify two expressions and returns `$true` if either one of them is `$true`.

PowerShell

```
if ( $age -le 13 -or $age -ge 55 )
```

Just like with the `-and` operator, the evaluation happens from left to right. Except that if the first part is `$true`, then the whole statement is `$true` and it doesn't process the rest of the expression.

Also make note of how the syntax works for these operators. You need two separate expressions. I have seen users try to do something like this `$value -eq 5 -or 6` without realizing their mistake.

-xor exclusive or

This one is a little unusual. `-xor` allows only one expression to evaluate to `$true`. So if both items are `$false` or both items are `$true`, then the whole expression is `$false`.

Another way to look at this is the expression is only `$true` when the results of the expression are different.

It's rare that anyone would ever use this logical operator and I can't think up a good example as to why I would ever use it.

Bitwise operators

Bitwise operators perform calculations on the bits within the values and produce a new value as the result. Teaching [bitwise operators](#) is beyond the scope of this article, but here is the list of them.

- `-band` binary AND
- `-bor` binary OR
- `-bxor` binary exclusive OR
- `-bnot` binary NOT
- `-shl` shift left
- `-shr` shift right

PowerShell expressions

We can use normal PowerShell inside the condition statement.

```
PowerShell
```

```
if ( Test-Path -Path $Path )
```

`Test-Path` returns `$true` or `$false` when it executes. This also applies to commands that return other values.

PowerShell

```
if ( Get-Process Notepad* )
```

It evaluates to `$true` if there's a returned process and `$false` if there isn't. It's perfectly valid to use pipeline expressions or other PowerShell statements like this:

PowerShell

```
if ( Get-Process | where Name -EQ Notepad )
```

These expressions can be combined with each other with the `-and` and `-or` operators, but you may have to use parenthesis to break them into subexpressions.

PowerShell

```
if ( (Get-Process) -and (Get-Service) )
```

Checking for `$null`

Having a no result or a `$null` value evaluates to `$false` in the `if` statement. When checking specifically for `$null`, it's a best practice to place the `$null` on the left-hand side.

PowerShell

```
if ( $null -eq $value )
```

There are quite a few nuances when dealing with `$null` values in PowerShell. If you're interested in diving deeper, I have an article about [everything you wanted to know about `\$null`](#).

Variable assignment within the condition

I almost forgot to add this one until [Prasoon Karunan V](#) reminded me of it.

PowerShell

```
if ($process=Get-Process notepad -ErrorAction Ignore) {$process} else
{$false}
```

Normally when you assign a value to a variable, the value isn't passed onto the pipeline or console. When you do a variable assignment in a sub expression, it does get passed on to the pipeline.

PowerShell

```
PS> $first = 1
PS> ($second = 2)
2
```

See how the `$first` assignment has no output and the `$second` assignment does?

When an assignment is done in an `if` statement, it executes just like the `$second` assignment above. Here is a clean example on how you could use it:

PowerShell

```
if ( $process = Get-Process Notepad* )
{
    $process | Stop-Process
}
```

If `$process` gets assigned a value, then the statement is `$true` and `$process` gets stopped.

Make sure you don't confuse this with `-eq` because this isn't an equality check. This is a more obscure feature that most people don't realize works this way.

Variable assignment from the scriptblock

You can also use the `if` statement scriptblock to assign a value to a variable.

PowerShell

```
$discount = if ( $age -ge 55 )
{
    Get-SeniorDiscount
}
elseif ( $age -le 13 )
{
    Get-ChildDiscount
```

```
    }
else
{
    0.00
}
```

Each script block is writing the results of the commands, or the value, as output. We can assign the result of the `if` statement to the `$discount` variable. That example could have just as easily assigned those values to the `$discount` variable directly in each scriptblock. I can't say that I use this with the `if` statement often, but I do have an example where I used this recently.

Alternate execution path

The `if` statement allows you to specify an action for not only when the statement is `$true`, but also for when it's `$false`. This is where the `else` statement comes into play.

else

The `else` statement is always the last part of the `if` statement when used.

PowerShell

```
if ( Test-Path -Path $Path -PathType Leaf )
{
    Move-Item -Path $Path -Destination $archivePath
}
else
{
    Write-Warning "$path doesn't exist or isn't a file."
}
```

In this example, we check the `$path` to make sure it's a file. If we find the file, we move it. If not, we write a warning. This type of branching logic is very common.

Nested if

The `if` and `else` statements take a script block, so we can place any PowerShell command inside them, including another `if` statement. This allows you to make use of much more complicated logic.

PowerShell

```
if ( Test-Path -Path $Path -PathType Leaf )
{
    Move-Item -Path $Path -Destination $archivePath
}
else
{
    if ( Test-Path -Path $Path )
    {
        Write-Warning "A file was required but a directory was found instead."
    }
    else
    {
        Write-Warning "$path could not be found."
    }
}
```

In this example, we test the happy path first and then take action on it. If that fails, we do another check and to provide more detailed information to the user.

elseif

We aren't limited to just a single conditional check. We can chain `if` and `else` statements together instead of nesting them by using the `elseif` statement.

```
PowerShell

if ( Test-Path -Path $Path -PathType Leaf )
{
    Move-Item -Path $Path -Destination $archivePath
}
elseif ( Test-Path -Path $Path )
{
    Write-Warning "A file was required but a directory was found instead."
}
else
{
    Write-Warning "$path could not be found."
}
```

The execution happens from the top to the bottom. The top `if` statement is evaluated first. If that is `$false`, then it moves down to the next `elseif` or `else` in the list. That last `else` is the default action to take if none of the others return `$true`.

switch

At this point, I need to mention the `switch` statement. It provides an alternate syntax for doing multiple comparisons with a value. With the `switch`, you specify an expression and that result gets compared with several different values. If one of those values match, the matching code block is executed. Take a look at this example:

```
PowerShell

$itemType = 'Role'
switch ( $ itemType )
{
    'Component'
    {
        'is a component'
    }
    'Role'
    {
        'is a role'
    }
    'Location'
    {
        'is a location'
    }
}
```

There three possible values that can match the `$itemType`. In this case, it matches with `Role`. I used a simple example just to give you some exposure to the `switch` operator. I talk more about [everything you ever wanted to know about the switch statement](#) in another article.

Array inline

I have a function called [Invoke-SnowSql](#) that launches an executable with several command-line arguments. Here is a clip from that function where I build the array of arguments.

```
PowerShell

$snowSqlParam = @(
    '--accountname', $Endpoint
    '--username', $Credential.UserName
    '--option', 'exit_on_error=true'
    '--option', 'output_format=csv'
    '--option', 'friendly=false'
    '--option', 'timing=false'
    if ($Debug)
    {
        '--option', 'log_level=DEBUG'
    }
```

```

if ($Path)
{
    '--filename', $Path
}
else
{
    '--query', $singleLineQuery
}
)

```

The `$Debug` and `$Path` variables are parameters on the function that are provided by the end user. I evaluate them inline inside the initialization of my array. If `$Debug` is true, then those values fall into the `$snowSqlParam` in the correct place. Same holds true for the `$Path` variable.

Simplify complex operations

It's inevitable that you run into a situation that has way too many comparisons to check and your `if` statement scrolls way off the right side of the screen.

PowerShell

```

$user = Get-ADUser -Identity $UserName
if ( $null -ne $user -and $user.Department -eq 'Finance' -and $user.Title -
match 'Senior' -and $user.HomeDrive -notlike '\\server\*' )
{
    # Do Something
}

```

They can be hard to read and that make you more prone to make mistakes. There are a few things we can do about that.

Line continuation

There some operators in PowerShell that let you wrap your command to the next line. The logical operators `-and` and `-or` are good operators to use if you want to break your expression into multiple lines.

PowerShell

```

if ($null -ne $user -and
    $user.Department -eq 'Finance' -and
    $user.Title -match 'Senior' -and
    $user.HomeDrive -notlike '\\server\*'
)

```

```
{  
    # Do Something  
}
```

There's still a lot going on there, but placing each piece on its own line makes a big difference. I generally use this when I get more than two comparisons or if I have to scroll to the right to read any of the logic.

Pre-calculating results

We can take that statement out of the `if` statement and only check the result.

```
PowerShell  
  
$needsSecureHomeDrive = $null -ne $user -and  
    $user.Department -eq 'Finance' -and  
    $user.Title -match 'Senior' -and  
    $user.HomeDrive -notlike '\\server*'  
  
if ( $needsSecureHomeDrive )  
{  
    # Do Something  
}
```

This just feels much cleaner than the previous example. You also are given an opportunity to use a variable name that explains what it's that you're really checking. This is also an example of self-documenting code that saves unnecessary comments.

Multiple if statements

We can break this up into multiple statements and check them one at a time. In this case, we use a flag or a tracking variable to combine the results.

```
PowerShell  
  
$skipUser = $false  
  
if( $null -eq $user )  
{  
    $skipUser = $true  
}  
  
if( $user.Department -ne 'Finance' )  
{  
    Write-Verbose "isn't in Finance department"  
    $skipUser = $true
```

```

}

if( $user.Title -match 'Senior' )
{
    Write-Verbose "Doesn't have Senior title"
    $skipUser = $true
}

if( $user.HomeDrive -like '\\server*' )
{
    Write-Verbose "Home drive already configured"
    $skipUser = $true
}

if ( -not $skipUser )
{
    # do something
}

```

I did have to invert the logic to make the flag logic work correctly. Each evaluation is an individual `if` statement. The advantage of this is that when you're debugging, you can tell exactly what the logic is doing. I was able to add much better verbosity at the same time.

The obvious downside is that it's so much more code to write. The code is more complex to look at as it takes a single line of logic and explodes it into 25 or more lines.

Using functions

We can also move all that validation logic into a function. Look at how clean this looks at a glance.

PowerShell

```

if ( Test-SecureDriveConfiguration -ADUser $user )
{
    # do something
}

```

You still have to create the function to do the validation, but it makes this code much easier to work with. It makes this code easier to test. In your tests, you can mock the call to `Test-ADDriveConfiguration` and you only need two tests for this function. One where it returns `$true` and one where it returns `$false`. Testing the other function is simpler because it's so small.

The body of that function could still be that one-liner we started with or the exploded logic that we used in the last section. This works well for both scenarios and allows you to easily change that implementation later.

Error handling

One important use of the `if` statement is to check for error conditions before you run into errors. A good example is to check if a folder already exists before you try to create it.

PowerShell

```
if ( -not (Test-Path -Path $folder) )
{
    New-Item -Type Directory -Path $folder
}
```

I like to say that if you expect an exception to happen, then it's not really an exception. So check your values and validate your conditions where you can.

If you want to dive a little more into actual exception handling, I have an article on [everything you ever wanted to know about exceptions](#).

Final words

The `if` statement is such a simple statement but is a fundamental piece of PowerShell. You will find yourself using this multiple times in almost every script you write. I hope you have a better understanding than you had before.

Everything you ever wanted to know about the switch statement

Article • 11/17/2022

Like many other languages, PowerShell has commands for controlling the flow of execution within your scripts. One of those statements is the `switch` statement and in PowerShell, it offers features that aren't found in other languages. Today, we take a deep dive into working with the PowerShell `switch`.

ⓘ Note

The [original version](#) of this article appeared on the blog written by [@KevinMarquette](#). The PowerShell team thanks Kevin for sharing this content with us. Please check out his blog at [PowerShellExplained.com](#).

The `if` statement

One of the first statements that you learn is the `if` statement. It lets you execute a script block if a statement is `$true`.

PowerShell

```
if ( Test-Path $Path )
{
    Remove-Item $Path
}
```

You can have much more complicated logic using `elseif` and `else` statements. Here is an example where I have a numeric value for day of the week and I want to get the name as a string.

PowerShell

```
$day = 3

if ( $day -eq 0 ) { $result = 'Sunday'      }
elseif ( $day -eq 1 ) { $result = 'Monday'     }
elseif ( $day -eq 2 ) { $result = 'Tuesday'    }
elseif ( $day -eq 3 ) { $result = 'Wednesday'  }
elseif ( $day -eq 4 ) { $result = 'Thursday'   }
elseif ( $day -eq 5 ) { $result = 'Friday'     }
elseif ( $day -eq 6 ) { $result = 'Saturday'  }
```

```
$result
```

Output

```
Wednesday
```

It turns out that this is a common pattern and there are many ways to deal with this. One of them is with a `switch`.

Switch statement

The `switch` statement allows you to provide a variable and a list of possible values. If the value matches the variable, then its scriptblock is executed.

PowerShell

```
$day = 3

switch ( $day )
{
    0 { $result = 'Sunday'      }
    1 { $result = 'Monday'      }
    2 { $result = 'Tuesday'     }
    3 { $result = 'Wednesday'   }
    4 { $result = 'Thursday'    }
    5 { $result = 'Friday'      }
    6 { $result = 'Saturday'    }
}

$result
```

Output

```
'Wednesday'
```

For this example, the value of `$day` matches one of the numeric values, then the correct name is assigned to `$result`. We're only doing a variable assignment in this example, but any PowerShell can be executed in those script blocks.

Assign to a variable

We can write that last example in another way.

PowerShell

```
$result = switch ( $day )
{
    0 { 'Sunday' }
    1 { 'Monday' }
    2 { 'Tuesday' }
    3 { 'Wednesday' }
    4 { 'Thursday' }
    5 { 'Friday' }
    6 { 'Saturday' }
}
```

We're placing the value on the PowerShell pipeline and assigning it to the `$result`. You can do this same thing with the `if` and `foreach` statements.

Default

We can use the `default` keyword to identify what should happen if there is no match.

PowerShell

```
$result = switch ( $day )
{
    0 { 'Sunday' }
    # ...
    6 { 'Saturday' }
    default { 'Unknown' }
}
```

Here we return the value `Unknown` in the default case.

Strings

I was matching numbers in those last examples, but you can also match strings.

PowerShell

```
$item = 'Role'

switch ( $item )
{
    Component
    {
        'is a component'
    }
}
```

```
Role
{
    'is a role'
}
Location
{
    'is a location'
}
}
```

Output

```
is a role
```

I decided not to wrap the `Component`, `Role` and `Location` matches in quotes here to highlight that they're optional. The `switch` treats those as a string in most cases.

Arrays

One of the cool features of the PowerShell `switch` is the way it handles arrays. If you give a `switch` an array, it processes each element in that collection.

PowerShell

```
$roles = @('WEB', 'Database')

switch ( $roles ) {
    'Database'   { 'Configure SQL' }
    'WEB'         { 'Configure IIS' }
    'FileServer' { 'Configure Share' }
}
```

Output

```
Configure IIS
Configure SQL
```

If you have repeated items in your array, then they're matched multiple times by the appropriate section.

PSItem

You can use the `$PSItem` or `$_` to reference the current item that was processed. When we do a simple match, `$PSItem` is the value that we're matching. I'll be performing some

advanced matches in the next section where this variable is used.

Parameters

A unique feature of the PowerShell `switch` is that it has a number of switch parameters that change how it performs.

-CaseSensitive

The matches aren't case-sensitive by default. If you need to be case-sensitive, you can use `-CaseSensitive`. This can be used in combination with the other switch parameters.

-Wildcard

We can enable wildcard support with the `-Wildcard` switch. This uses the same wildcard logic as the `-like` operator to do each match.

PowerShell

```
$Message = 'Warning, out of disk space'

switch -Wildcard ( $message )
{
    'Error*'
    {
        Write-Error -Message $Message
    }
    'Warning*'
    {
        Write-Warning -Message $Message
    }
    default
    {
        Write-Information $message
    }
}
```

Output

```
WARNING: Warning, out of disk space
```

Here we're processing a message and then outputting it on different streams based on the contents.

-Regex

The switch statement supports regex matches just like it does wildcards.

PowerShell

```
switch -Regex ( $message )
{
    '^Error'
    {
        Write-Error -Message $Message
    }
    '^Warning'
    {
        Write-Warning -Message $Message
    }
    default
    {
        Write-Information $message
    }
}
```

I have more examples of using regex in another article I wrote: [The many ways to use regex ↴](#).

-File

A little known feature of the switch statement is that it can process a file with the `-File` parameter. You use `-File` with a path to a file instead of giving it a variable expression.

PowerShell

```
switch -Wildcard -File $path
{
    'Error*'
    {
        Write-Error -Message $PSItem
    }
    'Warning*'
    {
        Write-Warning -Message $PSItem
    }
    default
    {
        Write-Output $PSItem
    }
}
```

It works just like processing an array. In this example, I combine it with wildcard matching and make use of the `$PSItem`. This would process a log file and convert it to warning and error messages depending on the regex matches.

Advanced details

Now that you're aware of all these documented features, we can use them in the context of more advanced processing.

Expressions

The `switch` can be on an expression instead of a variable.

PowerShell

```
switch ( ( Get-Service | where Status -EQ 'running' ).Name ) {...}
```

Whatever the expression evaluates to is the value used for the match.

Multiple matches

You may have already picked up on this, but a `switch` can match to multiple conditions. This is especially true when using `-Wildcard` or `-Regex` matches. You can add the same condition multiple times and all are triggered.

PowerShell

```
switch ( 'Word' )
{
    'word' { 'lower case word match' }
    'Word' { 'mixed case word match' }
    'WORD' { 'upper case word match' }
}
```

Output

```
lower case word match
mixed case word match
upper case word match
```

All three of these statements are fired. This shows that every condition is checked (in order). This holds true for processing arrays where each item checks each condition.

Continue

Normally, this is where I would introduce the `break` statement, but it's better that we learn how to use `continue` first. Just like with a `foreach` loop, `continue` continues onto the next item in the collection or exits the `switch` if there are no more items. We can rewrite that last example with `continue` statements so that only one statement executes.

PowerShell

```
switch ( 'Word' )
{
    'word'
    {
        'lower case word match'
        continue
    }
    'Word'
    {
        'mixed case word match'
        continue
    }
    'WORD'
    {
        'upper case word match'
        continue
    }
}
```

Output

```
lower case word match
```

Instead of matching all three items, the first one is matched and the switch continues to the next value. Because there are no values left to process, the switch exits. This next example is showing how a wildcard could match multiple items.

PowerShell

```
switch -Wildcard -File $path
{
    '*Error*'
    {
        Write-Error -Message $PSItem
        continue
    }
    '*Warning*'
    {
        Write-Warning -Message $PSItem
    }
}
```

```
        continue
    }
default
{
    Write-Output $PSItem
}
}
```

Because a line in the input file could contain both the word `Error` and `Warning`, we only want the first one to execute and then continue processing the file.

Break

A `break` statement exits the switch. This is the same behavior that `continue` presents for single values. The difference is shown when processing an array. `break` stops all processing in the switch and `continue` moves onto the next item.

PowerShell

```
$Messages = @(
    'Downloading update'
    'Ran into errors downloading file'
    'Error: out of disk space'
    'Sending email'
    '...'
)

switch -Wildcard ($Messages)
{
    'Error*'
    {
        Write-Error -Message $PSItem
        break
    }
    '*Error*'
    {
        Write-Warning -Message $PSItem
        continue
    }
    '*Warning*'
    {
        Write-Warning -Message $PSItem
        continue
    }
    default
    {
        Write-Output $PSItem
    }
}
```

Output

```
Downloading update
WARNING: Ran into errors downloading file
Write-Error -Message $PSItem : Error: out of disk space
+ CategoryInfo          : NotSpecified: (:) [Write-Error],
WriteErrorException
+ FullyQualifiedErrorCode : Microsoft.PowerShell.Commands.WriteErrorException
```

In this case, if we hit any lines that start with `Error` then we get an error and the switch stops. This is what that `break` statement is doing for us. If we find `Error` inside the string and not just at the beginning, we write it as a warning. We do the same thing for `Warning`. It's possible that a line could have both the word `Error` and `Warning`, but we only need one to process. This is what the `continue` statement is doing for us.

Break labels

The `switch` statement supports `break/continue` labels just like `foreach`.

PowerShell

```
:filelist foreach($path in $logs)
{
    :logFile switch -Wildcard -File $path
    {
        'Error*'
        {
            Write-Error -Message $PSItem
            break filelist
        }
        'Warning*'
        {
            Write-Error -Message $PSItem
            break logFile
        }
        default
        {
            Write-Output $PSItem
        }
    }
}
```

I personally don't like the use of break labels but I wanted to point them out because they're confusing if you've never seen them before. When you have multiple `switch` or `foreach` statements that are nested, you may want to break out of more than the inner most item. You can place a label on a `switch` that can be the target of your `break`.

Enum

PowerShell 5.0 gave us enums and we can use them in a switch.

```
PowerShell

enum Context {
    Component
    Role
    Location
}

$item = [Context]::Role

switch ( $item )
{
    Component
    {
        'is a component'
    }
    Role
    {
        'is a role'
    }
    Location
    {
        'is a location'
    }
}
```

```
Output
```

```
is a role
```

If you want to keep everything as strongly typed enums, then you can place them in parentheses.

```
PowerShell

switch ($item )
{
    ([Context]::Component)
    {
        'is a component'
    }
    ([Context]::Role)
    {
        'is a role'
    }
    ([Context]::Location)
```

```
{  
    'is a location'  
}  
}
```

The parentheses are needed here so that the switch doesn't treat the value `[Context]::Location` as a literal string.

ScriptBlock

We can use a scriptblock to perform the evaluation for a match if needed.

PowerShell

```
$age = 37  
  
switch ( $age )  
{  
    {$PSItem -le 18}  
    {  
        'child'  
    }  
    {$PSItem -gt 18}  
    {  
        'adult'  
    }  
}
```

Output

```
'adult'
```

This adds complexity and can make your `switch` hard to read. In most cases where you would use something like this it would be better to use `if` and `elseif` statements. I would consider using this if I already had a large switch in place and I needed two items to hit the same evaluation block.

One thing that I think helps with legibility is to place the scriptblock in parentheses.

PowerShell

```
switch ( $age )  
{  
    ($PSItem -le 18)  
    {  
        'child'  
    }  
}
```

```

    ($PSItem -gt 18)
{
    'adult'
}
}

```

It still executes the same way and gives a better visual break when quickly looking at it.

Regex \$Matches

We need to revisit regex to touch on something that isn't immediately obvious. The use of regex populates the `$Matches` variable. I do go into the use of `$Matches` more when I talk about [The many ways to use regex](#). Here is a quick sample to show it in action with named matches.

PowerShell

```

$message = 'my ssn is 123-23-3456 and credit card: 1234-5678-1234-5678'

switch -Regex ($message)
{
    '(?<SSN>\d\d\d-\d\d-\d\d\d\d\d)' {
        Write-Warning "message contains a SSN: $($Matches.SSN)"
    }
    '(?<CC>\d\d\d\d-\d\d\d\d-\d\d\d\d-\d\d\d\d\d\d)' {
        Write-Warning "message contains a credit card number:
 $($Matches.CC)"
    }
    '(?<Phone>\d\d\d-\d\d\d\d-\d\d\d\d\d)' {
        Write-Warning "message contains a phone number: $($Matches.Phone)"
    }
}

```

Output

```

WARNING: message may contain a SSN: 123-23-3456
WARNING: message may contain a credit card number: 1234-5678-1234-5678

```

\$null

You can match a `$null` value that doesn't have to be the default.

PowerShell

```
$values = '', 5, $null
switch ( $values )
{
    $null          { "Value '$_' is `$null" }
    '' -eq $_     { "Value '$_' is an empty string" }
    default       { "Value [$_] isn't an empty string or `$null" }
}
```

Output

```
Value '' is an empty string
Value [5] isn't an empty string or $null
Value '' is $null
```

When testing for an empty string in a `switch` statement, it's important to use the comparison statement as shown in this example instead of the raw value `''`. In a `switch` statement, the raw value `''` also matches `$null`. For example:

PowerShell

```
$values = '', 5, $null
switch ( $values )
{
    $null          { "Value '$_' is `$null" }
    ''            { "Value '$_' is an empty string" }
    default       { "Value [$_] isn't an empty string or `$null" }
}
```

Output

```
Value '' is an empty string
Value [5] isn't an empty string or $null
Value '' is $null
Value '' is an empty string
```

Also, be careful with empty returns from cmdlets. Cmdlets or pipelines that have no output are treated as an empty array that doesn't match anything, including the `default` case.

PowerShell

```
$file = Get-ChildItem NonExistantFile*
switch ( $file )
{
    $null      { '$file is $null' }
    default   { "`$file is type $($file.GetType().Name)" }
```

```
}
```

No matches

Constant expression

Lee Dailey pointed out that we can use a constant `$true` expression to evaluate `[bool]` items. Imagine if we have several boolean checks that need to happen.

PowerShell

```
$isVisible = $false
$isEnabled = $true
$isSecure = $true

switch ( $true )
{
    $isEnabled
    {
        'Do-Action'
    }
    $isVisible
    {
        'Show-Animation'
    }
    $isSecure
    {
        'Enable-AdminMenu'
    }
}
```

Output

```
Do-Action
Enabled-AdminMenu
```

This is a clean way to evaluate and take action on the status of several boolean fields. The cool thing about this is that you can have one match flip the status of a value that hasn't been evaluated yet.

PowerShell

```
$isVisible = $false
$isEnabled = $true
$isAdmin = $false

switch ( $true )
{
    $isEnabled
    {
```

```
{  
    'Do-Action'  
    $isVisible = $true  
}  
$isVisible  
{  
    'Show-Animation'  
}  
$isAdmin  
{  
    'Enable-AdminMenu'  
}  
}
```

Output

```
Do-Action  
Show-Animation
```

Setting `$isEnabled` to `$true` in this example makes sure that `$isVisible` is also set to `$true`. Then when `$isVisible` gets evaluated, its scriptblock is invoked. This is a bit counter-intuitive but is a clever use of the mechanics.

\$switch automatic variable

When the `switch` is processing its values, it creates an enumerator and calls it `$switch`. This is an automatic variable created by PowerShell and you can manipulate it directly.

PowerShell

```
$a = 1, 2, 3, 4  
  
switch($a) {  
    1 { [void]$switch.MoveNext(); $switch.Current }  
    3 { [void]$switch.MoveNext(); $switch.Current }  
}
```

This gives you the results of:

Output

```
2  
4
```

By moving the enumerator forward, the next item doesn't get processed by the `switch` but you can access that value directly. I would call it madness.

Other patterns

Hashtables

One of my most popular posts is the one I did on [hashtables](#). One of the use cases for a `hashtable` is to be a lookup table. That's an alternate approach to a common pattern that a `switch` statement is often addressing.

PowerShell

```
$day = 3

$lookup = @{
    0 = 'Sunday'
    1 = 'Monday'
    2 = 'Tuesday'
    3 = 'Wednesday'
    4 = 'Thursday'
    5 = 'Friday'
    6 = 'Saturday'
}

$lookup[$day]
```

Output

```
Wednesday
```

If I'm only using a `switch` as a lookup, I often use a `hashtable` instead.

Enum

PowerShell 5.0 introduced the `enum` and it's also an option in this case.

PowerShell

```
$day = 3

enum DayOfTheWeek {
    Sunday
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
```

```
}
```

```
[DayOfTheWeek]$day
```

Output

Wednesday

We could go all day looking at different ways to solve this problem. I just wanted to make sure you knew you had options.

Final words

The switch statement is simple on the surface but it offers some advanced features that most people don't realize are available. Stringing those features together makes this a powerful feature. I hope you learned something that you had not realized before.

Everything you wanted to know about exceptions

Article • 06/20/2024

Error handling is just part of life when it comes to writing code. We can often check and validate conditions for expected behavior. When the unexpected happens, we turn to exception handling. You can easily handle exceptions generated by other people's code or you can generate your own exceptions for others to handle.

ⓘ Note

The [original version](#) of this article appeared on the blog written by [@KevinMarquette](#). The PowerShell team thanks Kevin for sharing this content with us. Please check out his blog at [PowerShellExplained.com](#).

Basic terminology

We need to cover some basic terms before we jump into this one.

Exception

An Exception is like an event that is created when normal error handling can't deal with the issue. Trying to divide a number by zero or running out of memory are examples of something that creates an exception. Sometimes the author of the code you're using creates exceptions for certain issues when they happen.

Throw and Catch

When an exception happens, we say that an exception is thrown. To handle a thrown exception, you need to catch it. If an exception is thrown and it isn't caught by something, the script stops executing.

The call stack

The call stack is the list of functions that have called each other. When a function is called, it gets added to the stack or the top of the list. When the function exits or returns, it is removed from the stack.

When an exception is thrown, that call stack is checked in order for an exception handler to catch it.

Terminating and non-terminating errors

An exception is generally a terminating error. A thrown exception is either caught or it terminates the current execution. By default, a non-terminating error is generated by `Write-Error` and it adds an error to the output stream without throwing an exception.

I point this out because `Write-Error` and other non-terminating errors do not trigger the `catch`.

Swallowing an exception

This is when you catch an error just to suppress it. Do this with caution because it can make troubleshooting issues very difficult.

Basic command syntax

Here is a quick overview of the basic exception handling syntax used in PowerShell.

Throw

To create our own exception event, we throw an exception with the `throw` keyword.

```
PowerShell

function Start-Something
{
    throw "Bad thing happened"
}
```

This creates a runtime exception that is a terminating error. It's handled by a `catch` in a calling function or exits the script with a message like this.

```
PowerShell

PS> Start-Something

Bad thing happened
At line:1 char:1
+ throw "Bad thing happened"
+ ~~~~~~
```

```
+ CategoryInfo          : OperationStopped: (Bad thing happened:String)
[], RuntimeException
+ FullyQualifiedErrorId : Bad thing happened
```

Write-Error -ErrorAction Stop

I mentioned that `Write-Error` doesn't throw a terminating error by default. If you specify `-ErrorAction Stop`, `Write-Error` generates a terminating error that can be handled with a `catch`.

PowerShell

```
Write-Error -Message "Houston, we have a problem." -ErrorAction Stop
```

Thank you to Lee Dailey for reminding about using `-ErrorAction Stop` this way.

Cmdlet -ErrorAction Stop

If you specify `-ErrorAction Stop` on any advanced function or cmdlet, it turns all `Write-Error` statements into terminating errors that stop execution or that can be handled by a `catch`.

PowerShell

```
Start-Something -ErrorAction Stop
```

For more information about the `ErrorAction` parameter, see [about_CommonParameters](#).

For more information about the `$ErrorActionPreference` variable, see [about_Preference_Variables](#).

Try/Catch

The way exception handling works in PowerShell (and many other languages) is that you first `try` a section of code and if it throws an error, you can `catch` it. Here is a quick sample.

PowerShell

```
try
{
    Start-Something
}
```

```
catch
{
    Write-Output "Something threw an exception"
    Write-Output $_
}

try
{
    Start-Something -ErrorAction Stop
}
catch
{
    Write-Output "Something threw an exception or used Write-Error"
    Write-Output $_
}
```

The `catch` script only runs if there's a terminating error. If the `try` executes correctly, then it skips over the `catch`. You can access the exception information in the `catch` block using the `$_` variable.

Try/Finally

Sometimes you don't need to handle an error but still need some code to execute if an exception happens or not. A `finally` script does exactly that.

Take a look at this example:

PowerShell

```
$command = [System.Data.SqlClient.SqlCommand]::new(queryString, connection)
$command.Connection.Open()
$command.ExecuteNonQuery()
$command.Connection.Close()
```

Anytime you open or connect to a resource, you should close it. If the `ExecuteNonQuery()` throws an exception, the connection isn't closed. Here is the same code inside a `try/finally` block.

PowerShell

```
$command = [System.Data.SqlClient.SqlCommand]::new(queryString, connection)
try
{
    $command.Connection.Open()
    $command.ExecuteNonQuery()
}
finally
{
```

```
$command.Connection.Close()  
}
```

In this example, the connection is closed if there's an error. It also is closed if there's no error. The `finally` script runs every time.

Because you're not catching the exception, it still gets propagated up the call stack.

Try/Catch/Finally

It's perfectly valid to use `catch` and `finally` together. Most of the time you'll use one or the other, but you may find scenarios where you use both.

\$PSItem

Now that we got the basics out of the way, we can dig a little deeper.

Inside the `catch` block, there's an automatic variable (`$PSItem` or `$_`) of type `ErrorRecord` that contains the details about the exception. Here is a quick overview of some of the key properties.

For these examples, I used an invalid path in `ReadAllText` to generate this exception.

PowerShell

```
[System.IO.File]::ReadAllText( '\\\test\no\filefound.log')
```

PSItem.ToString()

This gives you the cleanest message to use in logging and general output. `ToString()` is automatically called if `$PSItem` is placed inside a string.

PowerShell

```
catch  
{  
    Write-Output "Ran into an issue: $($PSItem.ToString())"  
}  
  
catch  
{  
    Write-Output "Ran into an issue: $PSItem"  
}
```

\$PSItem.InvocationInfo

This property contains additional information collected by PowerShell about the function or script where the exception was thrown. Here is the `InvocationInfo` from the sample exception that I created.

```
PowerShell

PS> $PSItem.InvocationInfo | Format-List *

MyCommand          : Get-Resource
BoundParameters    : {}
UnboundArguments   : {}
ScriptLineNumber   : 5
OffsetInLine       : 5
ScriptName         : C:\blog\throwerror.ps1
Line               :      Get-Resource
PositionMessage    : At C:\blog\throwerror.ps1:5 char:5
                     +      Get-Resource
                     + ~~~~~
PSScriptRoot       : C:\blog
PSCommandPath      : C:\blog\throwerror.ps1
InvocationName     : Get-Resource
```

The important details here show the `ScriptName`, the `Line` of code and the `ScriptLineNumber` where the invocation started.

\$PSItem.ScriptStackTrace

This property shows the order of function calls that got you to the code where the exception was generated.

```
PowerShell

PS> $PSItem.ScriptStackTrace
at Get-Resource, C:\blog\throwerror.ps1: line 13
at Start-Something, C:\blog\throwerror.ps1: line 5
at <ScriptBlock>, C:\blog\throwerror.ps1: line 18
```

I'm only making calls to functions in the same script but this would track the calls if multiple scripts were involved.

\$PSItem.Exception

This is the actual exception that was thrown.

\$PSItem.Exception.Message

This is the general message that describes the exception and is a good starting point when troubleshooting. Most exceptions have a default message but can also be set to something custom when the exception is thrown.

PowerShell

```
PS> $PSItem.Exception.Message
```

```
Exception calling "ReadAllText" with "1" argument(s): "The network path was  
not found."
```

This is also the message returned when calling `$PSItem.ToString()` if there was not one set on the `ErrorRecord`.

\$PSItem.Exception.InnerException

Exceptions can contain inner exceptions. This is often the case when the code you're calling catches an exception and throws a different exception. The original exception is placed inside the new exception.

PowerShell

```
PS> $PSItem.Exception.InnerExceptionMessage  
The network path was not found.
```

I will revisit this later when I talk about re-throwing exceptions.

\$PSItem.Exception.StackTrace

This is the `StackTrace` for the exception. I showed a `ScriptStackTrace` above, but this one is for the calls to managed code.

Output

```
at System.IO.FileStream.Init(String path, FileMode mode, FileAccess access,  
Int32 rights, Boolean  
useRights, FileShare share, Int32 bufferSize, FileOptions options,  
SECURITY_ATTRIBUTES secAttrs,  
String msgPath, Boolean bFromProxy, Boolean useLongPath, Boolean checkHost)  
at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access,  
FileShare share, Int32  
bufferSize, FileOptions options, String msgPath, Boolean bFromProxy,  
Boolean useLongPath, Boolean
```

```
checkHost)
at System.IO.StreamReader..ctor(String path, Encoding encoding, Boolean
detectEncodingFromByteOrderMarks,
    Int32 bufferSize, Boolean checkHost)
at System.IO.File.InternalReadAllText(String path, Encoding encoding,
Boolean checkHost)
at CallSite.Target(Closure , CallSite , Type , String )
```

You only get this stack trace when the event is thrown from managed code. I'm calling a .NET Framework function directly so that is all we can see in this example. Generally when you're looking at a stack trace, you're looking for where your code stops and the system calls begin.

Working with exceptions

There is more to exceptions than the basic syntax and exception properties.

Catching typed exceptions

You can be selective with the exceptions that you catch. Exceptions have a type and you can specify the type of exception you want to catch.

PowerShell

```
try
{
    Start-Something -Path $path
}
catch [System.IO.FileNotFoundException]
{
    Write-Output "Could not find $path"
}
catch [System.IO.IOException]
{
    Write-Output "IO error with the file: $path"
}
```

The exception type is checked for each `catch` block until one is found that matches your exception. It's important to realize that exceptions can inherit from other exceptions. In the example above, `FileNotFoundException` inherits from `IOException`. So if the `IOException` was first, then it would get called instead. Only one catch block is invoked even if there are multiple matches.

If we had a `System.IO.PathTooLongException`, the `IOException` would match but if we had an `InsufficientMemoryException` then nothing would catch it and it would

propagate up the stack.

Catch multiple types at once

It's possible to catch multiple exception types with the same `catch` statement.

PowerShell

```
try
{
    Start-Something -Path $path -ErrorAction Stop
}
catch [System.IO.DirectoryNotFoundException],
[System.IO.FileNotFoundException]
{
    Write-Output "The path or file was not found: [$path]"
}
catch [System.IO.IOException]
{
    Write-Output "IO error with the file: [$path]"
}
```

Thank you Redditor `u/Sheppard_Ra` for suggesting this addition.

Throwing typed exceptions

You can throw typed exceptions in PowerShell. Instead of calling `throw` with a string:

PowerShell

```
throw "Could not find: $path"
```

Use an exception accelerator like this:

PowerShell

```
throw [System.IO.FileNotFoundException] "Could not find: $path"
```

But you have to specify a message when you do it that way.

You can also create a new instance of an exception to be thrown. The message is optional when you do this because the system has default messages for all built-in exceptions.

PowerShell

```
throw [System.IO.FileNotFoundException]::new()
throw [System.IO.FileNotFoundException]::new("Could not find path: $path")
```

If you're not using PowerShell 5.0 or higher, you must use the older `New-Object` approach.

PowerShell

```
throw (New-Object -TypeName System.IO.FileNotFoundException )
throw (New-Object -TypeName System.IO.FileNotFoundException -ArgumentList
"Could not find path: $path")
```

By using a typed exception, you (or others) can catch the exception by the type as mentioned in the previous section.

Write-Error -Exception

We can add these typed exceptions to `Write-Error` and we can still `catch` the errors by exception type. Use `Write-Error` like in these examples:

PowerShell

```
# with normal message
Write-Error -Message "Could not find path: $path" -Exception
([System.IO.FileNotFoundException]::new()) -ErrorAction Stop

# With message inside new exception
Write-Error -Exception ([System.IO.FileNotFoundException]::new("Could not
find path: $path")) -ErrorAction Stop

# Pre PS 5.0
Write-Error -Exception ([System.IO.FileNotFoundException]"Could not find
path: $path") -ErrorAction Stop

Write-Error -Message "Could not find path: $path" -Exception (New-Object -
TypeName System.IO.FileNotFoundException) -ErrorAction Stop
```

Then we can catch it like this:

PowerShell

```
catch [System.IO.FileNotFoundException]
{
    Write-Log $PSItem.ToString()
}
```

The big list of .NET exceptions

I compiled a master list with the help of the Reddit [r/PowerShell](#) community that contains hundreds of .NET exceptions to complement this post.

- [The big list of .NET exceptions ↗](#)

I start by searching that list for exceptions that feel like they would be a good fit for my situation. You should try to use exceptions in the base `System` namespace.

Exceptions are objects

If you start using a lot of typed exceptions, remember that they are objects. Different exceptions have different constructors and properties. If we look at the [FileNotFoundException](#) documentation for `System.IO.FileNotFoundException`, we see that we can pass in a message and a file path.

```
PowerShell
```

```
[System.IO.FileNotFoundException]::new("Could not find file", $path)
```

And it has a `FileName` property that exposes that file path.

```
PowerShell
```

```
catch [System.IO.FileNotFoundException]
{
    Write-Output $PSItem.Exception.FileName
}
```

You should consult the [.NET documentation](#) for other constructors and object properties.

Re-throwing an exception

If all you're going to do in your `catch` block is `throw` the same exception, then don't `catch` it. You should only `catch` an exception that you plan to handle or perform some action when it happens.

There are times where you want to perform an action on an exception but re-throw the exception so something downstream can deal with it. We could write a message or log the problem close to where we discover it but handle the issue further up the stack.

PowerShell

```
catch
{
    Write-Log $PSItem.ToString()
    throw $PSItem
}
```

Interestingly enough, we can call `throw` from within the `catch` and it re-throws the current exception.

PowerShell

```
catch
{
    Write-Log $PSItem.ToString()
    throw
}
```

We want to re-throw the exception to preserve the original execution information like source script and line number. If we throw a new exception at this point, it hides where the exception started.

Re-throwing a new exception

If you catch an exception but you want to throw a different one, then you should nest the original exception inside the new one. This allows someone down the stack to access it as the `$PSItem.Exception.InnerException`.

PowerShell

```
catch
{
    throw [System.MissingFieldException]::new('Could not access
field',$PSItem.Exception)
}
```

\$PSCmdlet.ThrowTerminatingError()

The one thing that I don't like about using `throw` for raw exceptions is that the error message points at the `throw` statement and indicates that line is where the problem is.

Output

```
Unable to find the specified file.  
At line:31 char:9  
+         throw [System.IO.FileNotFoundException]::new()  
+  
+             + CategoryInfo          : OperationStopped: () [],  
FileNotFoundException  
+ FullyQualifiedErrorId : Unable to find the specified file.
```

Having the error message tell me that my script is broken because I called `throw` on line 31 is a bad message for users of your script to see. It doesn't tell them anything useful.

Dexter Dhami pointed out that I can use `ThrowTerminatingError()` to correct that.

PowerShell

```
$PSCmdlet.ThrowTerminatingError(  
    [System.Management.Automation.ErrorRecord]::new(  
        ([System.IO.FileNotFoundException]"Could not find $Path"),  
        'My.ID',  
        [System.Management.Automation.ErrorCategory]::OpenError,  
        $MyObject  
    )  
)
```

If we assume that `ThrowTerminatingError()` was called inside a function called `Get-Resource`, then this is the error that we would see.

Output

```
Get-Resource : Could not find C:\Program Files (x86)\Reference  
Assemblies\Microsoft\Framework\.NETPortable\v4.6\System.IO.xml  
At line:6 char:5  
+     Get-Resource -Path $Path  
+  
+             + CategoryInfo          : OpenError: () [Get-Resource],  
FileNotFoundException  
+ FullyQualifiedErrorId : My.ID,Get-Resource
```

Do you see how it points to the `Get-Resource` function as the source of the problem? That tells the user something useful.

Because `$PSItem` is an `ErrorRecord`, we can also use `ThrowTerminatingError` this way to re-throw.

PowerShell

```
catch
{
    $PSCmdlet.ThrowTerminatingError($PSItem)
}
```

This changes the source of the error to the Cmdlet and hide the internals of your function from the users of your Cmdlet.

Try can create terminating errors

Kirk Munro points out that some exceptions are only terminating errors when executed inside a `try/catch` block. Here is the example he gave me that generates a divide by zero runtime exception.

```
PowerShell

function Start-Something { 1/(1-1) }
```

Then invoke it like this to see it generate the error and still output the message.

```
PowerShell

&{ Start-Something; Write-Output "We did it. Send Email" }
```

But by placing that same code inside a `try/catch`, we see something else happen.

```
PowerShell

try
{
    &{ Start-Something; Write-Output "We did it. Send Email" }
}
catch
{
    Write-Output "Notify Admin to fix error and send email"
}
```

We see the error become a terminating error and not output the first message. What I don't like about this one is that you can have this code in a function and it acts differently if someone is using a `try/catch`.

I have not ran into issues with this myself but it is corner case to be aware of.

\$PSCmdlet.ThrowTerminatingError() inside try/catch

One nuance of `$PSCmdlet.ThrowTerminatingError()` is that it creates a terminating error within your Cmdlet but it turns into a non-terminating error after it leaves your Cmdlet. This leaves the burden on the caller of your function to decide how to handle the error. They can turn it back into a terminating error by using `-ErrorAction Stop` or calling it from within a `try{...}catch{...}`.

Public function templates

One last take a way I had with my conversation with Kirk Munro was that he places a `try{...}catch{...}` around every `begin`, `process` and `end` block in all of his advanced functions. In those generic catch blocks, he has a single line using `$PSCmdlet.ThrowTerminatingError($PSItem)` to deal with all exceptions leaving his functions.

```
PowerShell

function Start-Something
{
    [CmdletBinding()]
    param()

    process
    {
        try
        {
            ...
        }
        catch
        {
            $PSCmdlet.ThrowTerminatingError($PSItem)
        }
    }
}
```

Because everything is in a `try` statement within his functions, everything acts consistently. This also gives clean errors to the end user that hides the internal code from the generated error.

Trap

I focused on the `try/catch` aspect of exceptions. But there's one legacy feature I need to mention before we wrap this up.

A `trap` is placed in a script or function to catch all exceptions that happen in that scope. When an exception happens, the code in the `trap` is executed and then the normal code continues. If multiple exceptions happen, then the trap is called over and over.

PowerShell

```
trap
{
    Write-Log $PSItem.ToString()
}

throw [System.Exception]::new('first')
throw [System.Exception]::new('second')
throw [System.Exception]::new('third')
```

I personally never adopted this approach but I can see the value in admin or controller scripts that log any and all exceptions, then still continue to execute.

Closing remarks

Adding proper exception handling to your scripts not only make them more stable, but also makes it easier for you to troubleshoot those exceptions.

I spent a lot of time talking `throw` because it is a core concept when talking about exception handling. PowerShell also gave us `Write-Error` that handles all the situations where you would use `throw`. So don't think that you need to be using `throw` after reading this.

Now that I have taken the time to write about exception handling in this detail, I'm going to switch over to using `Write-Error -Stop` to generate errors in my code. I'm also going to take Kirk's advice and make `ThrowTerminatingError` my goto exception handler for every function.

Everything you wanted to know about \$null

Article • 06/11/2024

The PowerShell `$null` often appears to be simple but it has a lot of nuances. Let's take a close look at `$null` so you know what happens when you unexpectedly run into a `$null` value.

ⓘ Note

The [original version](#) of this article appeared on the blog written by [@KevinMarquette](#). The PowerShell team thanks Kevin for sharing this content with us. Please check out his blog at [PowerShellExplained.com](#).

What is NULL?

You can think of NULL as an unknown or empty value. A variable is NULL until you assign a value or an object to it. This can be important because there are some commands that require a value and generate errors if the value is NULL.

PowerShell \$null

`$null` is an automatic variable in PowerShell used to represent NULL. You can assign it to variables, use it in comparisons and use it as a place holder for NULL in a collection.

PowerShell treats `$null` as an object with a value of NULL. This is different than what you may expect if you come from another language.

Examples of \$null

Anytime you try to use a variable that you have not initialized, the value is `$null`. This is one of the most common ways that `$null` values sneak into your code.

PowerShell

```
PS> $null -eq $undefinedVariable  
True
```

If you happen to mistype a variable name then PowerShell sees it as a different variable and the value is `$null`.

The other way you find `$null` values is when they come from other commands that don't give you any results.

PowerShell

```
PS> function Get-Nothing {}  
PS> $value = Get-Nothing  
PS> $null -eq $value  
True
```

Impact of `$null`

`$null` values impact your code differently depending on where they show up.

In strings

If you use `$null` in a string, then it's a blank value (or empty string).

PowerShell

```
PS> $value = $null  
PS> Write-Output "'The value is $value'"  
'The value is '
```

This is one of the reasons that I like to place brackets around variables when using them in log messages. It's even more important to identify the edges of your variable values when the value is at the end of the string.

PowerShell

```
PS> $value = $null  
PS> Write-Output "The value is [$value]"  
The value is []
```

This makes empty strings and `$null` values easy to spot.

In numeric equation

When a `$null` value is used in a numeric equation then your results are invalid if they don't give an error. Sometimes the `$null` evaluates to `0` and other times it makes the whole result `$null`. Here is an example with multiplication that gives 0 or `$null` depending on the order of the values.

PowerShell

```
PS> $null * 5
PS> $null -eq ( $null * 5 )
True

PS> 5 * $null
0
PS> $null -eq ( 5 * $null )
False
```

In place of a collection

A collection allows you use an index to access values. If you try to index into a collection that is actually `null`, you get this error: `Cannot index into a null array`.

PowerShell

```
PS> $value = $null
PS> $value[10]
Cannot index into a null array.
At line:1 char:1
+ $value[10]
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [], RuntimeException
+ FullyQualifiedErrorId : NullArray
```

If you have a collection but try to access an element that is not in the collection, you get a `$null` result.

PowerShell

```
$array = @('one','two','three')
$null -eq $array[100]
True
```

In place of an object

If you try to access a property or sub property of an object that doesn't have the specified property, you get a `$null` value like you would for an undefined variable. It doesn't matter if the variable is `$null` or an actual object in this case.

```
PowerShell
```

```
PS> $null -eq $undefined.Some.Fake.Property
True

PS> $date = Get-Date
PS> $null -eq $date.Some.Fake.Property
True
```

Method on a null-valued expression

Calling a method on a `$null` object throws a `RuntimeException`.

```
PowerShell
```

```
PS> $value = $null
PS> $value.ToString()
You cannot call a method on a null-valued expression.
At line:1 char:1
+ $value.ToString()
+ ~~~~~
+ CategoryInfo          : InvalidOperationException: () [], RuntimeException
+ FullyQualifiedErrorId : InvokeMethodOnNull
```

Whenever I see the phrase `You cannot call a method on a null-valued expression` then the first thing I look for are places where I am calling a method on a variable without first checking it for `$null`.

Checking for `$null`

You may have noticed that I always place the `$null` on the left when checking for `$null` in my examples. This is intentional and accepted as a PowerShell best practice. There are some scenarios where placing it on the right doesn't give you the expected result.

Look at this next example and try to predict the results:

```
PowerShell
```

```
if ( $value -eq $null )
{
    'The array is $null'
```

```
}
```

```
if ( $value -ne $null )
```

```
{
```

```
    'The array is not $null'
```

```
}
```

If I do not define `$value`, the first one evaluates to `$true` and our message is `The array is $null`. The trap here is that it's possible to create a `$value` that allows both of them to be `$false`

PowerShell

```
$value = @( $null )
```

In this case, the `$value` is an array that contains a `$null`. The `-eq` checks every value in the array and returns the `$null` that is matched. This evaluates to `$false`. The `-ne` returns everything that doesn't match `$null` and in this case there are no results (This also evaluates to `$false`). Neither one is `$true` even though it looks like one of them should be.

Not only can we create a value that makes both of them evaluate to `$false`, it's possible to create a value where they both evaluate to `$true`. Mathias Jessen (@IISResetMe) has a [good post](#) that dives into that scenario.

PSScriptAnalyzer and VSCode

The [PSScriptAnalyzer](#) module has a rule that checks for this issue called `PSPossibleIncorrectComparisonWithNull`.

PowerShell

```
PS> Invoke-ScriptAnalyzer ./myscript.ps1
```

RuleName	Message
-----	-----
PSPossibleIncorrectComparisonWithNull	<code>\$null</code> should be on the left side of equality comparisons.

Because VS Code uses the PSScriptAnalyzer rules too, it also highlights or identifies this as a problem in your script.

Simple if check

A common way that people check for a non-\$null value is to use a simple `if()` statement without the comparison.

PowerShell

```
if ( $value )
{
    Do-Something
}
```

If the value is `$null`, this evaluates to `$false`. This is easy to read, but be careful that it's looking for exactly what you're expecting it to look for. I read that line of code as:

If `$value` has a value.

But that's not the whole story. That line is actually saying:

If `$value` is not `$null` or `0` or `$false` or an empty string or an empty array.

Here is a more complete sample of that statement.

PowerShell

```
if ( $null -ne $value -and
    $value -ne 0 -and
    $value -ne '' -and
    ($value -isnot [array] -or $value.Length -ne 0) -and
    $value -ne $false )
{
    Do-Something
}
```

It's perfectly OK to use a basic `if` check as long as you remember those other values count as `$false` and not just that a variable has a value.

I ran into this issue when refactoring some code a few days ago. It had a basic property check like this.

PowerShell

```
if ( $object.Property )
{
    $object.Property = $value
}
```

I wanted to assign a value to the object property only if it existed. In most cases, the original object had a value that would evaluate to `$true` in the `if` statement. But I ran into an issue where the value was occasionally not getting set. I debugged the code and found that the object had the property but it was a blank string value. This prevented it from ever getting updated with the previous logic. So I added a proper `$null` check and everything worked.

PowerShell

```
if ( $null -ne $object.Property )  
{  
    $object.Property = $value  
}
```

It's little bugs like these that are hard to spot and make me aggressively check values for `$null`.

\$null.Count

If you try to access a property on a `$null` value, that the property is also `$null`. The `Count` property is the exception to this rule.

PowerShell

```
PS> $value = $null  
PS> $value.Count  
0
```

When you have a `$null` value, then the `Count` is `0`. This special property is added by PowerShell.

[PSCustomObject] Count

Almost all objects in PowerShell have that `Count` property. One important exception is the `[pscustomobject]` in Windows PowerShell 5.1 (This is fixed in PowerShell 6.0). It doesn't have a `Count` property so you get a `$null` value if you try to use it. I call this out here so that you don't try to use `Count` instead of a `$null` check.

Running this example on Windows PowerShell 5.1 and PowerShell 6.0 gives you different results.

PowerShell

```
$value = [pscustomobject]@{Name='MyObject'}
if ( $value.Count -eq 1 )
{
    "We have a value"
}
```

Enumerable null

There is one special type of `$null` that acts differently than the others. I am going to call it the enumerable null but it's really a [System.Management.Automation.Internal.AutomationNull](#). This enumerable null is the one you get as the result of a function or script block that returns nothing (a void result).

PowerShell

```
PS> function Get-Nothing {}
PS> $nothing = Get-Nothing
PS> $null -eq $nothing
True
```

If you compare it with `$null`, you get a `$null` value. When used in an evaluation where a value is required, the value is always `$null`. But if you place it inside an array, it's treated the same as an empty array.

PowerShell

```
PS> $containEmpty = @( () )
PS> $containNothing = @($nothing)
PS> $containNull = @($null)

PS> $containEmpty.Count
0
PS> $containNothing.Count
0
PS> $containNull.Count
1
```

You can have an array that contains one `$null` value and its `Count` is `1`. But if you place an empty array inside an array then it's not counted as an item. The count is `0`.

If you treat the enumerable null like a collection, then it's empty.

If you pass in an enumerable null to a function parameter that isn't strongly typed, PowerShell coerces the enumerable null into a `$null` value by default. This means inside

the function, the value is treated as `$null` instead of the `System.Management.Automation.Internal.AutomationNull` type.

Pipeline

The primary place you see the difference is when using the pipeline. You can pipe a `$null` value but not an enumerable null value.

PowerShell

```
PS> $null | ForEach-Object{ Write-Output 'NULL Value' }
'NULL Value'
PS> $nothing | ForEach-Object{ Write-Output 'No Value' }
```

Depending on your code, you should account for the `$null` in your logic.

Either check for `$null` first

- Filter out null on the pipeline (`... | where {$null -ne $_} | ...`)
- Handle it in the pipeline function

foreach

One of my favorite features of `foreach` is that it doesn't enumerate over a `$null` collection.

PowerShell

```
foreach ( $node in $null )
{
    #skipped
}
```

This saves me from having to `$null` check the collection before I enumerate it. If you have a collection of `$null` values, the `$node` can still be `$null`.

The `foreach` started working this way with PowerShell 3.0. If you happen to be on an older version, then this is not the case. This is one of the important changes to be aware of when back-porting code for 2.0 compatibility.

Value types

Technically, only reference types can be `$null`. But PowerShell is very generous and allows for variables to be any type. If you decide to strongly type a value type, it cannot be `$null`. PowerShell converts `$null` to a default value for many types.

PowerShell

```
PS> [int]$number = $null
PS> $number
0

PS> [bool]$boolean = $null
PS> $boolean
False

PS> [string]$string = $null
PS> $string -eq ''
True
```

There are some types that do not have a valid conversion from `$null`. These types generate a `Cannot convert null to type` error.

PowerShell

```
PS> [datetime]$date = $null
Cannot convert null to type "System.DateTime".
At line:1 char:1
+ [datetime]$date = $null
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [],
ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException
```

Function parameters

Using a strongly typed values in function parameters is very common. We generally learn to define the types of our parameters even if we tend not to define the types of other variables in our scripts. You may already have some strongly typed variables in your functions and not even realize it.

PowerShell

```
function Do-Something
{
    param(
        [string] $Value
```

```
)  
}
```

As soon as you set the type of the parameter as a `string`, the value can never be `$null`. It's common to check if a value is `$null` to see if the user provided a value or not.

PowerShell

```
if ( $null -ne $Value ){...}
```

`$Value` is an empty string `''` when no value is provided. Use the automatic variable `$PSBoundParameters.Value` instead.

PowerShell

```
if ( $null -ne $PSBoundParameters.Value ){...}
```

`$PSBoundParameters` only contains the parameters that were specified when the function was called. You can also use the `ContainsKey` method to check for the property.

PowerShell

```
if ( $PSBoundParameters.ContainsKey('Value') ){...}
```

IsNotNullOrEmpty

If the value is a string, you can use a static string function to check if the value is `$null` or an empty string at the same time.

PowerShell

```
if ( -not [string]::IsNullOrEmpty( $value ) ){...}
```

I find myself using this often when I know the value type should be a string.

When I `$null` check

I am a defensive scripter. Anytime I call a function and assign it to a variable, I check it for `$null`.

PowerShell

```
$userList = Get-ADUser kevmar
if ($null -ne $userList){...}
```

I much prefer using `if` or `foreach` over using `try/catch`. Don't get me wrong, I still use `try/catch` a lot. But if I can test for an error condition or an empty set of results, I can allow my exception handling be for true exceptions.

I also tend to check for `$null` before I index into a value or call methods on an object. These two actions fail for a `$null` object so I find it important to validate them first. I already covered those scenarios earlier in this post.

No results scenario

It's important to know that different functions and commands handle the no results scenario differently. Many PowerShell commands return the enumerable null and an error in the error stream. But others throw exceptions or give you a status object. It's still up to you to know how the commands you use deal with the no results and error scenarios.

Initializing to `$null`

One habit that I have picked up is initializing all my variables before I use them. You are required to do this in other languages. At the top of my function or as I enter a `foreach` loop, I define all the values that I'm using.

Here is a scenario that I want you to take a close look at. It's an example of a bug I had to chase down before.

PowerShell

```
function Do-Something
{
    foreach ( $node in 1..6 )
    {
        try
        {
            $result = Get-Something -Id $node
        }
        catch
        {
            Write-Verbose "[ $result ] not valid"
        }

        if ( $null -ne $result )
```

```
    {
        Update-Something $result
    }
}
```

The expectation here is that `Get-Something` returns either a result or an enumerable null. If there's an error, we log it. Then we check to make sure we got a valid result before processing it.

The bug hiding in this code is when `Get-Something` throws an exception and doesn't assign a value to `$result`. It fails before the assignment so we don't even assign `$null` to the `$result` variable. `$result` still contains the previous valid `$result` from other iterations. `Update-Something` to execute multiple times on the same object in this example.

I set `$result` to `$null` right inside the `foreach` loop before I use it to mitigate this issue.

PowerShell

```
foreach ( $node in 1..6 )
{
    $result = $null
    try
    {
        ...
    }
```

Scope issues

This also helps mitigate scoping issues. In that example, we assign values to `$result` over and over in a loop. But because PowerShell allows variable values from outside the function to bleed into the scope of the current function, initializing them inside your function mitigates bugs that can be introduced that way.

An uninitialized variable in your function is not `$null` if it's set to a value in a parent scope. The parent scope could be another function that calls your function and uses the same variable names.

If I take that same `Do-something` example and remove the loop, I would end up with something that looks like this example:

PowerShell

```

function Invoke-Something
{
    $result = 'ParentScope'
    Do-Something
}

function Do-Something
{
    try
    {
        $result = Get-Something -Id $node
    }
    catch
    {
        Write-Verbose "[{$result}] not valid"
    }

    if ( $null -ne $result )
    {
        Update-Something $result
    }
}

```

If the call to `Get-Something` throws an exception, then my `$null` check finds the `$result` from `Invoke-Something`. Initializing the value inside your function mitigates this issue.

Naming variables is hard and it's common for an author to use the same variable names in multiple functions. I know I use `$node`, `$result`, `$data` all the time. So it would be very easy for values from different scopes to show up in places where they should not be.

Redirect output to `$null`

I have been talking about `$null` values for this entire article but the topic is not complete if I didn't mention redirecting output to `$null`. There are times when you have commands that output information or objects that you want to suppress. Redirecting output to `$null` does that.

Out-Null

The `Out-Null` command is the built-in way to redirect pipeline data to `$null`.

PowerShell

```
New-Item -Type Directory -Path $path | Out-Null
```

Assign to \$null

You can assign the results of a command to `$null` for the same effect as using `Out-Null`.

```
PowerShell
```

```
$null = New-Item -Type Directory -Path $path
```

Because `$null` is a constant value, you can never overwrite it. I don't like the way it looks in my code but it often performs faster than `Out-Null`.

Redirect to \$null

You can also use the redirection operator to send output to `$null`.

```
PowerShell
```

```
New-Item -Type Directory -Path $path > $null
```

If you're dealing with command-line executables that output on the different streams. You can redirect all output streams to `$null` like this:

```
PowerShell
```

```
git status *> $null
```

Summary

I covered a lot of ground on this one and I know this article is more fragmented than most of my deep dives. That is because `$null` values can pop up in many different places in PowerShell and all the nuances are specific to where you find it. I hope you walk away from this with a better understanding of `$null` and an awareness of the more obscure scenarios you may run into.

Everything you wanted to know about ShouldProcess

Article • 09/24/2024

PowerShell functions have several features that greatly improve the way users interact with them. One important feature that is often overlooked is `-WhatIf` and `-Confirm` support and it's easy to add to your functions. In this article, we dive deep into how to implement this feature.

ⓘ Note

The [original version](#) of this article appeared on the blog written by [@KevinMarquette](#). The PowerShell team thanks Kevin for sharing this content with us. Please check out his blog at [PowerShellExplained.com](#).

This is a simple feature you can enable in your functions to provide a safety net for the users that need it. There's nothing scarier than running a command that you know can be dangerous for the first time. The option to run it with `-WhatIf` can make a big difference.

CommonParameters

Before we look at implementing these [common parameters](#), I want to take a quick look at how they're used.

Using -WhatIf

When a command supports the `-WhatIf` parameter, it allows you to see what the command would have done instead of making changes. It's a good way to test out the impact of a command, especially before you do something destructive.

PowerShell

```
PS C:\temp> Get-ChildItem
Directory: C:\temp

Mode                LastWriteTime         Length Name
----                -----          ----
-a----   4/19/2021 8:59 AM            0 importantfile.txt
-a----   4/19/2021 8:58 AM            0 myfile1.txt
-a----   4/19/2021 8:59 AM            0 myfile2.txt
```

```
PS C:\temp> Remove-Item -Path .\myfile1.txt -WhatIf
What if: Performing the operation "Remove File" on target
"C:\Temp\myfile1.txt".
```

If the command correctly implements `ShouldProcess`, it should show you all the changes that it would have made. Here is an example using a wildcard to delete multiple files.

PowerShell

```
PS C:\temp> Remove-Item -Path * -WhatIf
What if: Performing the operation "Remove File" on target
"C:\Temp\myfile1.txt".
What if: Performing the operation "Remove File" on target
"C:\Temp\myfile2.txt".
What if: Performing the operation "Remove File" on target
"C:\Temp\importantfile.txt".
```

Using -Confirm

Commands that support `-WhatIf` also support `-Confirm`. This gives you a chance to confirm an action before performing it.

PowerShell

```
PS C:\temp> Remove-Item .\myfile1.txt -Confirm

Confirm
Are you sure you want to perform this action?
Performing the operation "Remove File" on target "C:\Temp\myfile1.txt".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):
```

In this case, you have multiple options that allow you to continue, skip a change, or stop the script. The help prompt describes each of those options like this.

Output

```
Y - Continue with only the next step of the operation.
A - Continue with all the steps of the operation.
N - Skip this operation and proceed with the next operation.
L - Skip this operation and all subsequent operations.
S - Pause the current pipeline and return to the command prompt. Type "exit"
to resume the pipeline.
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):
```

Localization

This prompt is localized in PowerShell so the language changes based on the language of your operating system. This is one more thing that PowerShell manages for you.

Switch parameters

Let's take quick moment to look at ways to pass a value to a switch parameter. The main reason I call this out is that you often want to pass parameter values to functions you call.

The first approach is a specific parameter syntax that can be used for all parameters but you mostly see it used for switch parameters. You specify a colon to attach a value to the parameter.

```
PowerShell  
  
Remove-Item -Path:* -WhatIf:$true
```

You can do the same with a variable.

```
PowerShell  
  
$DoWhatIf = $true  
Remove-Item -Path * -WhatIf:$DoWhatIf
```

The second approach is to use a hashtable to splat the value.

```
PowerShell  
  
$RemoveSplat = @{  
    Path = '*'  
    WhatIf = $true  
}  
Remove-Item @RemoveSplat
```

If you're new to hashtables or splatting, I have another article on that covers [everything you wanted to know about hashtables](#).

SupportsShouldProcess

The first step to enable `-WhatIf` and `-Confirm` support is to specify `SupportsShouldProcess` in the `CmdletBinding` of your function.

PowerShell

```
function Test-ShouldProcess {  
    [CmdletBinding(SupportsShouldProcess)]  
    param()  
    Remove-Item .\myfile1.txt  
}
```

By specifying `SupportsShouldProcess` in this way, we can now call our function with `-WhatIf` (or `-Confirm`).

PowerShell

```
PS> Test-ShouldProcess -WhatIf  
What if: Performing the operation "Remove File" on target  
"C:\Temp\myfile1.txt".
```

Notice that I did not create a parameter called `-WhatIf`. Specifying `SupportsShouldProcess` automatically creates it for us. When we specify the `-WhatIf` parameter on `Test-ShouldProcess`, some things we call also perform `-WhatIf` processing.

ⓘ Note

When you use `SupportsShouldProcess`, PowerShell doesn't add the `$WhatIf` variable to the function. You don't need to check the value of `$WhatIf` because the `ShouldProcess()` method takes care of that for you.

Trust but verify

There's some danger here trusting that everything you call inherits `-WhatIf` values. For the rest of the examples, I'm going to assume that it doesn't work and be very explicit when making calls to other commands. I recommend that you do the same.

PowerShell

```
function Test-ShouldProcess {  
    [CmdletBinding(SupportsShouldProcess)]  
    param()  
    Remove-Item .\myfile1.txt -WhatIf:$WhatIfPreference  
}
```

I will revisit the nuances much later once you have a better understanding of all the pieces in play.

\$PSCmdlet.ShouldProcess

The method that allows you to implement `SupportsShouldProcess` is `$PSCmdlet.ShouldProcess`. You call `$PSCmdlet.ShouldProcess(...)` to see if you should process some logic and PowerShell takes care of the rest. Let's start with an example:

```
PowerShell

function Test-ShouldProcess {
    [CmdletBinding(SupportsShouldProcess)]
    param()

    $file = Get-ChildItem './myfile1.txt'
    if($PSCmdlet.ShouldProcess($file.Name)){
        $file.Delete()
    }
}
```

The call to `$PSCmdlet.ShouldProcess($file.Name)` checks for the `-WhatIf` (and `-Confirm` parameter) then handles it accordingly. The `-WhatIf` causes `ShouldProcess` to output a description of the change and return `$false`:

```
PowerShell

PS> Test-ShouldProcess -WhatIf
What if: Performing the operation "Test-ShouldProcess" on target
"myfile1.txt".
```

A call using `-Confirm` pauses the script and prompts the user with the option to continue. It returns `$true` if the user selected `Y`.

```
PowerShell

PS> Test-ShouldProcess -Confirm
Confirm
Are you sure you want to perform this action?
Performing the operation "Test-ShouldProcess" on target "myfile1.txt".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):
```

An awesome feature of `$PSCmdlet.ShouldProcess` is that it doubles as verbose output. I depend on this often when implementing `ShouldProcess`.

PowerShell

```
PS> Test-ShouldProcess -Verbose  
VERBOSE: Performing the operation "Test-ShouldProcess" on target  
"myfile1.txt".
```

Overloads

There are a few different overloads for `$PSCmdlet.ShouldProcess` with different parameters for customizing the messaging. We already saw the first one in the example above. Let's take a closer look at it.

PowerShell

```
function Test-ShouldProcess {  
    [CmdletBinding(SupportsShouldProcess)]  
    param()  
  
    if($PSCmdlet.ShouldProcess('TARGET')){  
        # ...  
    }  
}
```

This produces output that includes both the function name and the target (value of the parameter).

PowerShell

```
What if: Performing the operation "Test-ShouldProcess" on target "TARGET".
```

Specifying a second parameter as the operation uses the operation value instead of the function name in the message.

PowerShell

```
## $PSCmdlet.ShouldProcess('TARGET','OPERATION')  
What if: Performing the operation "OPERATION" on target "TARGET".
```

The next option is to specify three parameters to fully customize the message. When three parameters are used, the first one is the entire message. The second two parameters are still used in the `-Confirm` message output.

PowerShell

```
## $PSCmdlet.ShouldProcess('MESSAGE','TARGET','OPERATION')
What if: MESSAGE
```

Quick parameter reference

Just in case you came here only to figure out what parameters you should use, here is a quick reference showing how the parameters change the message in the different `-WhatIf` scenarios.

PowerShell

```
## $PSCmdlet.ShouldProcess('TARGET')
What if: Performing the operation "FUNCTION_NAME" on target "TARGET".

## $PSCmdlet.ShouldProcess('TARGET','OPERATION')
What if: Performing the operation "OPERATION" on target "TARGET".

## $PSCmdlet.ShouldProcess('MESSAGE','TARGET','OPERATION')
What if: MESSAGE
```

I tend to use the one with two parameters.

ShouldProcessReason

We have a fourth overload that's more advanced than the others. It allows you to get the reason `ShouldProcess` was executed. I'm only adding this here for completeness because we can just check if `$WhatIfPreference` is `$true` instead.

PowerShell

```
$reason = ''
if($PSCmdlet.ShouldProcess('MESSAGE','TARGET','OPERATION',[ref]$reason)){
    Write-Output "Some Action"
}
$reason
```

We have to pass the `$reason` variable into the fourth parameter as a reference variable with `[ref]`. `ShouldProcess` populates `$reason` with the value `None` or `WhatIf`. I didn't say this was useful and I have had no reason to ever use it.

Where to place it

You use `ShouldProcess` to make your scripts safer. So you use it when your scripts are making changes. I like to place the `$PSCmdlet.ShouldProcess` call as close to the change as possible.

PowerShell

```
## general logic and variable work
if ($PSCmdlet.ShouldProcess('TARGET', 'OPERATION')){
    # Change goes here
}
```

If I'm processing a collection of items, I call it for each item. So the call gets placed inside the `foreach` loop.

PowerShell

```
foreach ($node in $collection){
    # general logic and variable work
    if ($PSCmdlet.ShouldProcess($node, 'OPERATION')){
        # Change goes here
    }
}
```

The reason why I place `ShouldProcess` tightly around the change, is that I want as much code to execute as possible when `-WhatIf` is specified. I want the setup and validation to run if possible so the user gets to see those errors.

I also like to use this in my Pester tests that validate my projects. If I have a piece of logic that is hard to mock in pester, I can often wrap it in `ShouldProcess` and call it with `-WhatIf` in my tests. It's better to test some of your code than none of it.

\$WhatIfPreference

The first preference variable we have is `$WhatIfPreference`. This is `$false` by default. If you set it to `$true` then your function executes as if you specified `-WhatIf`. If you set this in your session, all commands perform `-WhatIf` execution.

When you call a function with `-WhatIf`, the value of `$WhatIfPreference` gets set to `$true` inside the scope of your function.

ConfirmImpact

Most of my examples are for `-WhatIf` but everything so far also works with `-Confirm` to prompt the user. You can set the `ConfirmImpact` of the function to high and it prompts the user as if it was called with `-Confirm`.

PowerShell

```
function Test-ShouldProcess {
    [CmdletBinding(
        SupportsShouldProcess,
        ConfirmImpact = 'High'
    )]
    param()

    if ($PSCmdlet.ShouldProcess('TARGET')){
        Write-Output "Some Action"
    }
}
```

This call to `Test-ShouldProcess` is performing the `-Confirm` action because of the `High` impact.

PowerShell

```
PS> Test-ShouldProcess

Confirm
Are you sure you want to perform this action?
Performing the operation "Test-ShouldProcess" on target "TARGET".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"): y
Some Action
```

The obvious issue is that now it's harder to use in other scripts without prompting the user. In this case, we can pass a `$false` to `-Confirm` to suppress the prompt.

PowerShell

```
PS> Test-ShouldProcess -Confirm:$false
Some Action
```

I'll cover how to add `-Force` support in a later section.

\$ConfirmPreference

`$ConfirmPreference` is an automatic variable that controls when `ConfirmImpact` asks you to confirm execution. Here are the possible values for both `$ConfirmPreference` and

`ConfirmImpact`.

- `High`
- `Medium`
- `Low`
- `None`

With these values, you can specify different levels of impact for each function. If you have `$ConfirmPreference` set to a value higher than `ConfirmImpact`, then you aren't prompted to confirm execution.

By default, `$ConfirmPreference` is set to `High` and `ConfirmImpact` is `Medium`. If you want your function to automatically prompt the user, set your `ConfirmImpact` to `High`. Otherwise set it to `Medium` if its destructive and use `Low` if the command is always safe run in production. If you set it to `none`, it doesn't prompt even if `-Confirm` was specified (but it still gives you `-WhatIf` support).

When calling a function with `-Confirm`, the value of `$ConfirmPreference` gets set to `Low` inside the scope of your function.

Suppressing nested confirm prompts

The `$ConfirmPreference` can get picked up by functions that you call. This can create scenarios where you add a confirm prompt and the function you call also prompts the user.

What I tend to do is specify `-Confirm:$false` on the commands that I call when I have already handled the prompting.

PowerShell

```
function Test-ShouldProcess {
    [CmdletBinding(SupportsShouldProcess)]
    param()

    $file = Get-ChildItem './myfile1.txt'
    if($PSCmdlet.ShouldProcess($file.Name)){
        Remove-Item -Path $file.FullName -Confirm:$false
    }
}
```

This brings us back to an earlier warning: There are nuances as to when `-WhatIf` is not passed to a function and when `-Confirm` passes to a function. I promise I'll get back to this later.

\$PSCmdlet.ShouldContinue

If you need more control than `ShouldProcess` provides, you can trigger the prompt directly with `ShouldContinue`. `ShouldContinue` ignores `$ConfirmPreference`, `ConfirmImpact`, `-Confirm`, `$WhatIfPreference`, and `-WhatIf` because it prompts every time it's executed.

At a quick glance, it's easy to confuse `ShouldProcess` and `ShouldContinue`. I tend to remember to use `ShouldProcess` because the parameter is called `SupportsShouldProcess` in the `CmdletBinding`. You should use `ShouldProcess` in almost every scenario. That is why I covered that method first.

Let's take a look at `ShouldContinue` in action.

```
PowerShell

function Test-ShouldContinue {
    [CmdletBinding()]
    param()

    if($PSCmdlet.ShouldContinue('TARGET', 'OPERATION')){
        Write-Output "Some Action"
    }
}
```

This provides us a simpler prompt with fewer options.

```
PowerShell

Test-ShouldContinue

Second
TARGET
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"):
```

The biggest issue with `ShouldContinue` is that it requires the user to run it interactively because it always prompts the user. You should always be building tools that can be used by other scripts. The way you do this is by implementing `-Force`. I'll revisit this idea later.

Yes to all

This is automatically handled with `ShouldProcess` but we have to do a little more work for `ShouldContinue`. There's a second method overload where we have to pass in a few

values by reference to control the logic.

PowerShell

```
function Test-ShouldContinue {
    [CmdletBinding()]
    param()

    $collection = 1..5
    $yesToAll = $false
    $noToAll = $false

    foreach($target in $collection) {

        $continue = $PSCmdlet.ShouldContinue(
            "TARGET_$target",
            'OPERATION',
            [ref]$yesToAll,
            [ref]$noToAll
        )

        if ($continue){
            Write-Output "Some Action [$target]"
        }
    }
}
```

I added a `foreach` loop and a collection to show it in action. I pulled the `ShouldContinue` call out of the `if` statement to make it easier to read. Calling a method with four parameters starts to get a little ugly, but I tried to make it look as clean as I could.

Implementing -Force

`ShouldProcess` and `ShouldContinue` need to implement `-Force` in different ways. The trick to these implementations is that `ShouldProcess` should always get executed, but `ShouldContinue` should not get executed if `-Force` is specified.

ShouldProcess -Force

If you set your `ConfirmImpact` to `high`, the first thing your users are going to try is to suppress it with `-Force`. That's the first thing I do anyway.

PowerShell

```
Test-ShouldProcess -Force
Error: Test-ShouldProcess: A parameter cannot be found that matches
```

```
parameter name 'force'.
```

If you recall from the `ConfirmImpact` section, they actually need to call it like this:

PowerShell

```
Test-ShouldProcess -Confirm:$false
```

Not everyone realizes they need to do that and `-Force` doesn't suppress `ShouldContinue`. So we should implement `-Force` for the sanity of our users. Take a look at this full example here:

PowerShell

```
function Test-ShouldProcess {
    [CmdletBinding(
        SupportsShouldProcess,
        ConfirmImpact = 'High'
    )]
    param(
        [switch]$Force
    )

    if ($Force -and -not $PSBoundParameters.ContainsKey('Confirm')) {
        $ConfirmPreference = 'None'
    }

    if ($PSCmdlet.ShouldProcess('TARGET')) {
        Write-Output "Some Action"
    }
}
```

We add our own `-Force` switch as a parameter. The `-Confirm` parameter is automatically added when using `SupportsShouldProcess` in the `CmdletBinding`. However, when you use `SupportsShouldProcess`, PowerShell doesn't add the `$Confirm` variable to the function. If you are running in Strict Mode and try to use the `$Confirm` variable before it has been defined, you get an error. To avoid the error you can use `$PSBoundParameters` to test if the parameter was passed by the user.

PowerShell

```
if ($Force -and -not $PSBoundParameters.ContainsKey('Confirm')) {
    $ConfirmPreference = 'None'
}
```

If the user specifies `-Force` we set `$ConfirmPreference` to `None` in the local scope. If the user also specifies `-Confirm` then `ShouldProcess()` honors the values of the `-Confirm` parameter.

PowerShell

```
if ($PSCmdlet.ShouldProcess('TARGET')){
    Write-Output "Some Action"
}
```

If someone specifies both `-Force` and `-WhatIf`, then `-WhatIf` needs to take priority. This approach preserves `-WhatIf` processing because `ShouldProcess` always gets executed.

Don't add a test for the `$Force` value inside the `if` statement with the `ShouldProcess`. That is an anti-pattern for this specific scenario even though that's what I show you in the next section for `ShouldContinue`.

ShouldContinue -Force

This is the correct way to implement `-Force` with `ShouldContinue`.

PowerShell

```
function Test-ShouldContinue {
    [CmdletBinding()]
    param(
        [switch]$Force
    )

    if($Force -or $PSCmdlet.ShouldContinue('TARGET', 'OPERATION')){
        Write-Output "Some Action"
    }
}
```

By placing the `$Force` to the left of the `-or` operator, it gets evaluated first. Writing it this way short circuits the execution of the `if` statement. If `$Force` is `$true`, then the `ShouldContinue` is not executed.

PowerShell

```
PS> Test-ShouldContinue -Force
Some Action
```

We don't have to worry about `-Confirm` or `-WhatIf` in this scenario because they're not supported by `ShouldContinue`. This is why it needs to be handled differently than `ShouldProcess`.

Scope issues

Using `-WhatIf` and `-Confirm` are supposed to apply to everything inside your functions and everything they call. They do this by setting `$WhatIfPreference` to `$true` or setting `$ConfirmPreference` to `Low` in the local scope of the function. When you call another function, calls to `ShouldProcess` use those values.

This actually works correctly most of the time. Anytime you call built-in cmdlet or a function in your same scope, it works. It also works when you call a script or a function in a script module from the console.

The one specific place where it doesn't work is when a script or a script module calls a function in another script module. This may not sound like a big problem, but most of the modules you create or pull from the PSGallery are script modules.

The core issue is that script modules do not inherit the values for `$WhatIfPreference` or `$ConfirmPreference` (and several others) when called from functions in other script modules.

The best way to summarize this as a general rule is that this works correctly for binary modules and never trust it to work for script modules. If you aren't sure, either test it or just assume it doesn't work correctly.

I personally feel this is very dangerous because it creates scenarios where you add `-WhatIf` support to multiple modules that work correctly in isolation, but fail to work correctly when they call each other.

We do have a GitHub RFC working to get this issue fixed. See [Propagate execution preferences beyond script module scope ↗](#) for more details.

In closing

I have to look up how to use `ShouldProcess` every time I need to use it. It took me a long time to distinguish `ShouldProcess` from `ShouldContinue`. I almost always need to look up what parameters to use. So don't worry if you still get confused from time to time. This article will be here when you need it. I'm sure I will reference it often myself.

If you liked this post, please share your thoughts with me on Twitter using the link below. I always like hearing from people that get value from my content.

Visualize parameter binding

Article • 05/20/2024

Parameter binding is the process that PowerShell uses to determine which parameter set is being used and to associate (bind) values to the parameters of a command. These values can come from the command line and the pipeline.

The parameter binding process starts by binding named and positional command-line arguments. After binding command-line arguments, PowerShell tries to bind any pipeline input. There are two ways that values are bound from the pipeline. Parameters that accept pipeline input have one or both of the following attributes:

- [ValueFromPipeline](#) - The value from the pipeline is bound to the parameter based on its type. The type of the argument must match the type of the parameter.
- [ValueFromPipelineByPropertyName](#) - The value from the pipeline is bound to the parameter based on its name. The object in the pipeline must have a property that matches the name of the parameter or one of its aliases. The type of the property must match or be convertible to the type of the parameter.

For more information about parameter binding, see [about_Parameter_Binding](#).

Use [Trace-Command](#) to visualize parameter binding

Troubleshooting parameter binding issues can be challenging. You can use the [Trace-Command](#) cmdlet to visualize the parameter binding process.

Consider the following scenario. You have a directory with two text files, `file1.txt` and `[file2].txt`.

```
PowerShell

PS> Get-ChildItem

Directory: D:\temp\test\binding

Mode          LastWriteTime      Length Name
----          -----          ---- 
-a---  5/17/2024 12:59 PM        0 [file2].txt
-a---  5/17/2024 12:59 PM        0 file1.txt
```

You want to delete the files by passing the filenames, through the pipeline, to the `Remove-Item` cmdlet.

PowerShell

```
PS> 'file1.txt', '[file2].txt' | Remove-Item
PS> Get-ChildItem

Directory: D:\temp\test\binding

Mode                LastWriteTime         Length Name
----                -----          ----- 
-a---      5/17/2024 12:59 PM            0 [file2].txt
```

Notice that `Remove-Item` only deleted `file1.txt` and not `[file2].txt`. The filename includes square brackets, which is treated as a wildcard expression. Using `Trace-Command`, you can see that the filename is being bound to the `Path` parameter of `Remove-Item`.

PowerShell

```
Trace-Command -PSHost -Name ParameterBinding -Expression {
    '[file2].txt' | Remove-Item
}
```

The output from `Trace-Command` can be verbose. Each line of output is prefixed with a timestamp and trace provider information. For the output of this example, the prefix information has been removed to make it easier to read.

Output

```
BIND NAMED cmd line args [Remove-Item]
BIND POSITIONAL cmd line args [Remove-Item]
BIND cmd line args to DYNAMIC parameters.
    DYNAMIC parameter object:
[Microsoft.PowerShell.Commands.FileSystemProviderRemoveItemDynamicParameters
]
MANDATORY PARAMETER CHECK on cmdlet [Remove-Item]
CALLING BeginProcessing
BIND PIPELINE object to parameters: [Remove-Item]
    PIPELINE object TYPE = [System.String]
    RESTORING pipeline parameter's original values
    Parameter [Path] PIPELINE INPUT ValueFromPipeline NO COERCION
    BIND arg [[file2].txt] to parameter [Path]
        Binding collection parameter Path: argument type [String], parameter
        type [System.String[]],
            collection type Array, element type [System.String], no
            coerceElementType
```

```

Creating array with element type [System.String] and 1 elements
Argument type String is not IList, treating this as scalar
Adding scalar element of type String to array position 0
BIND arg [System.String[]] to param [Path] SUCCESSFUL
Parameter [Credential] PIPELINE INPUT ValueFromPipelineByPropertyName NO
COERCION
Parameter [Credential] PIPELINE INPUT ValueFromPipelineByPropertyName NO
COERCION
Parameter [Credential] PIPELINE INPUT ValueFromPipelineByPropertyName
WITH COERCION
Parameter [Credential] PIPELINE INPUT ValueFromPipelineByPropertyName
WITH COERCION
MANDATORY PARAMETER CHECK on cmdlet [Remove-Item]
CALLING ProcessRecord
CALLING EndProcessing

```

Using `Get-Help`, you can see that the `Path` parameter of `Remove-Item` accepts string objects from the pipeline `ByValue` or `ByPropertyName`. `LiteralPath` accepts string objects from the pipeline `ByPropertyName`.

PowerShell

```

PS> Get-Help Remove-Item -Parameter Path, LiteralPath

-Path <System.String[]>
    Specifies a path of the items being removed. Wildcard characters are
    permitted.

    Required?          true
    Position?         0
    Default value     None
    Accept pipeline input? True (ByPropertyName, ByValue)
    Accept wildcard characters? true


-LiteralPath <System.String[]>
    Specifies a path to one or more locations. The value of LiteralPath is
    used exactly as it's
        typed. No characters are interpreted as wildcards. If the path includes
        escape characters,
            enclose it in single quotation marks. Single quotation marks tell
        PowerShell not to interpret
            any characters as escape sequences.

    Required?          true
    Position?         named
    Default value     None
    Accept pipeline input? True (ByPropertyName)
    Accept wildcard characters? false

```

The output of `Trace-Command` shows that parameter binding starts by binding command-line parameters followed by the pipeline input. You can see that `Remove-Item` receives a string object from the pipeline. That string object is bound to the **Path** parameter.

```
BIND PIPELINE object to parameters: [Remove-Item]
PIPELINE object TYPE = [System.String]
RESTORING pipeline parameter's original values
Parameter [Path] PIPELINE INPUT ValueFromPipeline NO COERCION
BIND arg [[file2].txt] to parameter [Path]
...
BIND arg [System.String[]] to param [Path] SUCCESSFUL
```

Since the **Path** parameter accepts wildcard characters, the square brackets represent a wildcard expression. However, that expression doesn't match any files in the directory. You need to use the **LiteralPath** parameter to specify the exact path to the file.

`Get-Command` shows that the **LiteralPath** parameter accepts input from the pipeline `ByPropertyName` or `ByValue`. And, that it has two aliases, `PSPath` and `LP`.

PowerShell

```
PS> (Get-Command Remove-Item).Parameters.LiteralPath.Attributes |
>> Select-Object ValueFrom*, Alias* | Format-List

ValueFromPipeline          : False
ValueFromPipelineByPropertyName : True
ValueFromRemainingArguments   : False

AliasNames : {PSPath, LP}
```

In this next example, `Get-Item` is used to retrieve a **FileInfo** object. That object has a property named **PSPath**.

PowerShell

```
PS> Get-Item *.txt | Select-Object PSPath

PSPath
-----
Microsoft.PowerShell.Core\FileSystem::D:\temp\test\binding\[file2].txt
```

The **FileInfo** object is then passed to `Remove-Item`.

PowerShell

```
Trace-Command -PSHost -Name ParameterBinding -Expression {
    Get-Item *.txt | Remove-Item
}
```

For the output of this example, the prefix information has been removed and separated to show parameter binding for both commands.

In this output, you can see that `Get-Item` binds the positional parameter value `*.txt` to the **Path** parameter.

Output

```
BIND NAMED cmd line args [Get-Item]
BIND POSITIONAL cmd line args [Get-Item]
    BIND arg [*.txt] to parameter [Path]
        Binding collection parameter Path: argument type [String], parameter
        type [System.String[]],
            collection type Array, element type [System.String], no
        coerceElementType
            Creating array with element type [System.String] and 1 elements
            Argument type String is not IList, treating this as scalar
            Adding scalar element of type String to array position 0
            BIND arg [System.String[]} to param [Path] SUCCESSFUL
BIND cmd line args to DYNAMIC parameters.
    DYNAMIC parameter object:
[Microsoft.PowerShell.Commands.FileSystemProviderGetItemDynamicParameters]
MANDATORY PARAMETER CHECK on cmdlet [Get-Item]
```

In the trace output for parameter binding, you can see that `Remove-Item` receives a **FileInfo** object from the pipeline. Since a **FileInfo** object isn't a **String** object, it can't be bound to the **Path** parameter.

The **PSPath** property of the **FileInfo** object matches an alias for the **LiteralPath** parameter. **PSPath** is also a **String** object, so it can be bound to the **LiteralPath** parameter without type coercion.

Output

```
BIND NAMED cmd line args [Remove-Item]
BIND POSITIONAL cmd line args [Remove-Item]
BIND cmd line args to DYNAMIC parameters.
    DYNAMIC parameter object:
[Microsoft.PowerShell.Commands.FileSystemProviderRemoveItemDynamicParameters
]
MANDATORY PARAMETER CHECK on cmdlet [Remove-Item]
CALLING BeginProcessing
CALLING BeginProcessing
CALLING ProcessRecord
```

```
BIND PIPELINE object to parameters: [Remove-Item]
PIPELINE object TYPE = [System.IO.FileInfo]
RESTORING pipeline parameter's original values
Parameter [Path] PIPELINE INPUT ValueFromPipeline NO COERCION
BIND arg [D:\temp\test\binding\[file2].txt] to parameter [Path]
    Binding collection parameter Path: argument type [FileInfo],
parameter type [System.String[]],
        collection type Array, element type [System.String], no
coerceElementType
        Creating array with element type [System.String] and 1 elements
        Argument type FileInfo is not IList, treating this as scalar
        BIND arg [D:\temp\test\binding\[file2].txt] to param [Path]
SKIPPED
    Parameter [Credential] PIPELINE INPUT
ValueFromPipelineByPropertyName NO COERCION
    Parameter [Path] PIPELINE INPUT ValueFromPipelineByPropertyName NO
COERCION
    Parameter [Credential] PIPELINE INPUT
ValueFromPipelineByPropertyName NO COERCION
    Parameter [LiteralPath] PIPELINE INPUT
ValueFromPipelineByPropertyName NO COERCION
    BIND arg
[Microsoft.PowerShell.Core\FileSystem::D:\temp\test\binding\[file2].txt] to
parameter [LiteralPath]
        Binding collection parameter LiteralPath: argument type
[String], parameter type [System.String[]],
        collection type Array, element type [System.String], no
coerceElementType
        Creating array with element type [System.String] and 1 elements
        Argument type String is not IList, treating this as scalar
        Adding scalar element of type String to array position 0
        BIND arg [System.String[]} to param [LiteralPath] SUCCESSFUL
    Parameter [Credential] PIPELINE INPUT
ValueFromPipelineByPropertyName WITH COERCION
    MANDATORY PARAMETER CHECK on cmdlet [Remove-Item]
    CALLING ProcessRecord
    CALLING EndProcessing
    CALLING EndProcessing
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Writing Progress across multiple threads with `ForEach-Object -Parallel`

Article • 11/17/2022

Starting in PowerShell 7.0, the ability to work in multiple threads simultaneously is possible using the `Parallel` parameter in the `ForEach-Object` cmdlet. Monitoring the progress of these threads can be a challenge though. Normally, you can monitor the progress of a process using `Write-Progress`. However, since PowerShell uses a separate runspace for each thread when using `Parallel`, reporting the progress back to the host isn't as straight forward as normal use of `Write-Progress`.

Using a synced hashtable to track progress

When writing the progress from multiple threads, tracking becomes difficult because when running parallel processes in PowerShell, each process has its own runspace. To get around this, you can use a [synchronized hashtable](#). A synced hashtable is a thread safe data structure that can be modified by multiple threads simultaneously without throwing an error.

Set up

One of the downsides to this approach is it takes a, somewhat, complex set up to ensure everything runs without error.

```
PowerShell

$dataset = @(
    @{
        Id      = 1
        Wait   = 3..10 | Get-Random | ForEach-Object {$_*100}
    }
    @{
        Id      = 2
        Wait   = 3..10 | Get-Random | ForEach-Object {$_*100}
    }
    @{
        Id      = 3
        Wait   = 3..10 | Get-Random | ForEach-Object {$_*100}
    }
    @{
        Id      = 4
        Wait   = 3..10 | Get-Random | ForEach-Object {$_*100}
    }
)
```

```

@{
    Id    = 5
    Wait = 3..10 | Get-Random | ForEach-Object {$_.*100}
}
)

# Create a hashtable for process.
# Keys should be ID's of the processes
$origin = @{}
$dataset | ForEach-Object {$origin.($_.Id) = @{}}

# Create synced hashtable
$sync = [System.Collections.Hashtable]::Synchronized($origin)

```

This section creates three different data structures, for three different purposes.

The `$dataset` variable stores an array of hashtables that is used to coordinate the next steps without the risk of being modified. If an object collection is modified while iterating through the collection, PowerShell throws an error. You must keep the object collection in the loop separate from the objects being modified. The `Id` key in each hashtable is the identifier for a mock process. The `Wait` key simulates the workload of each mock process being tracked.

The `$origin` variable stores a nested hashtable with each key being one of the mock process id's. Then, it is used to hydrate the synchronized hashtable stored in the `$sync` variable. The `$sync` variable is responsible for reporting the progress back to the parent runspace, which displays the progress.

Running the processes

This section runs the multi-threaded processes and creates some of the output used to display progress.

PowerShell

```

$job = $dataset | ForEach-Object -ThrottleLimit 3 -AsJob -Parallel {
    $syncCopy = $Using:sync
    $process = $syncCopy.$($PSItem.Id)

    $process.Id = $PSItem.Id
    $process.Activity = "Id $($PSItem.Id) starting"
    $process.Status = "Processing"

    # Fake workload start up that takes x amount of time to complete
    Start-Sleep -Milliseconds ($PSItem.Wait*5)

    # Process. update activity
    $process.Activity = "Id $($PSItem.Id) processing"

```

```

foreach ($percent in 1..100)
{
    # Update process on status
    $process.Status = "Handling $percent/100"
    $process.PercentComplete = (($percent / 100) * 100)

    # Fake workload that takes x amount of time to complete
    Start-Sleep -Milliseconds $PSItem.Wait
}

# Mark process as completed
$process.Completed = $true
}

```

The mock processes are sent to `ForEach-Object` and started as jobs. The `ThrottleLimit` is set to `3` to highlight running multiple processes in a queue. The jobs are stored in the `$job` variable and allows us to know when all the processes have finished later on.

When using the `Using:` statement to reference a parent scope variable in PowerShell, you can't use expressions to make it dynamic. For example, if you tried to create the `$process` variable like this, `$process = $Using:sync.$($PSItem.Id)`, you would get an error stating you can't use expressions there. So, we create the `$syncCopy` variable to be able to reference and modify the `$sync` variable without the risk of it failing.

Next, we build out a hashtable to represent the progress of the process currently in the loop using the `$process` variable by referencing the synchronized hashtable keys. The `Activity` and the `Status` keys are used as parameter values for `Write-Progress` to display the status of a given mock process in the next section.

The `foreach` loop is just a way to simulate the process working and is randomized based on the `$dataSet` `Wait` attribute to set `Start-Sleep` using milliseconds. How you calculate the progress of your process may vary.

Displaying the progress of multiple processes

Now that the mock processes are running as jobs, we can start to write the processes progress to the PowerShell window.

```

PowerShell

while($job.State -eq 'Running')
{
    $sync.Keys | ForEach-Object {
        # If key is not defined, ignore
        if(![string]::IsNullOrEmpty($sync.$_.Keys))
        {

```

```

        # Create parameter hashtable to splat
        $param = $sync.$_

        # Execute Write-Progress
        Write-Progress @param
    }
}

# Wait to refresh to not overload gui
Start-Sleep -Seconds 0.1
}

```

The `$job` variable contains the parent **job** and has a child **job** for each of the mock processes. While any of the child jobs are still running, the parent job **State** will remain "Running". This allows us to use the `while` loop to continually update the progress of every process until all processes are finished.

Within the while loop, we loop through each of the keys in the `$sync` variable. Since this is a synchronized hashtable, it is constantly updated but can still be accessed without throwing any errors.

There is a check to ensure that the process being reported is actually running using the `IsNullOrEmpty()` method. If the process hasn't been started, the loop won't report on it and move on to the next until it gets to a process that has been started. If the process is started, the hashtable from the current key is used to splat the parameters to `Write-Progress`.

Full example

PowerShell

```

# Example workload
$dataset = @(
    @{
        Id      = 1
        Wait   = 3..10 | Get-Random | ForEach-Object {$_ *100}
    }
    @{
        Id      = 2
        Wait   = 3..10 | Get-Random | ForEach-Object {$_ *100}
    }
    @{
        Id      = 3
        Wait   = 3..10 | Get-Random | ForEach-Object {$_ *100}
    }
    @{
        Id      = 4
        Wait   = 3..10 | Get-Random | ForEach-Object {$_ *100}
    }
)

```

```

}
@{
    Id    = 5
    Wait = 3..10 | Get-Random | ForEach-Object {$_*100}
}
)

# Create a hashtable for process.
# Keys should be ID's of the processes
$origin = @{}
$dataset | ForEach-Object {$origin.($_.Id) = @{}}

# Create synced hashtable
$sync = [System.Collections.Hashtable]::Synchronized($origin)

$job = $dataset | ForEach-Object -ThrottleLimit 3 -AsJob -Parallel {
    $syncCopy = $Using:sync
    $process = $syncCopy.$($PSItem.Id)

    $process.Id = $PSItem.Id
    $process.Activity = "Id $($PSItem.Id) starting"
    $process.Status = "Processing"

    # Fake workload start up that takes x amount of time to complete
    Start-Sleep -Milliseconds ($PSItem.Wait*5)

    # Process. update activity
    $process.Activity = "Id $($PSItem.Id) processing"
    foreach ($percent in 1..100)
    {
        # Update process on status
        $process.Status = "Handling $percent/100"
        $process.PercentComplete = (($percent / 100) * 100)

        # Fake workload that takes x amount of time to complete
        Start-Sleep -Milliseconds $PSItem.Wait
    }

    # Mark process as completed
    $process.Completed = $true
}

while($job.State -eq 'Running')
{
    $sync.Keys | ForEach-Object {
        # If key is not defined, ignore
        if(![string]::IsNullOrEmpty($sync.$_.Keys))
        {
            # Create parameter hashtable to splat
            $param = $sync.$_

            # Execute Write-Progress
            Write-Progress @param
        }
    }
}

```

```
# Wait to refresh to not overload gui
Start-Sleep -Seconds 0.1
}
```

Related Links

- [about_Jobs](#)
- [about_Scopes](#)
- [about_Splatting](#)

Add Credential support to PowerShell functions

Article • 11/17/2022

ⓘ Note

The [original version](#) of this article appeared on the blog written by [@joshduffney](#). This article has been edited for inclusion on this site. The PowerShell team thanks Josh for sharing this content with us. Please check out his blog at [duffney.io](#).

This article shows you how to add credential parameters to PowerShell functions and why you'd want to. A credential parameter is to allow you to run the function or cmdlet as a different user. The most common use is to run the function or cmdlet as an elevated user account.

For example, the cmdlet `New-ADUser` has a **Credential** parameter, which you could provide domain admin credentials to create an account in a domain. Assuming your normal account running the PowerShell session doesn't have that access already.

Creating credential object

The **PSCredential** object represents a set of security credentials such as a user name and password. The object can be passed as a parameter to a function that runs as the user account in that credential object. There are a few ways that you can create a credential object. The first way to create a credential object is to use the PowerShell cmdlet `Get-Credential`. When you run without parameters, it prompts you for a username and password. Or you can call the cmdlet with some optional parameters.

To specify the domain name and username ahead of time you can use either the **Credential** or **UserName** parameters. When you use the **UserName** parameter, you're also required to provide a **Message** value. The code below demonstrates using the cmdlet. You can also store the credential object in a variable so that you can use the credential multiple times. In the example below, the credential object is stored in the variable `$cred`.

PowerShell

```
$Cred = Get-Credential  
$Cred = Get-Credential -Credential domain\user  
$Cred = Get-Credential -UserName domain\user -Message 'Enter Password'
```

Sometimes, you can't use the interactive method of creating credential objects shown in the previous example. Most automation tools require a non-interactive method. To create a credential without user interaction, create a secure string containing the password. Then pass the secure string and user name to the `System.Management.Automation.PSCredential()` method.

Use the following command to create a secure string containing the password:

PowerShell

```
ConvertTo-SecureString "MyPlainTextPassword" -AsPlainText -Force
```

Both the `AsPlainText` and `Force` parameters are required. Without those parameters, you receive a message warning that you shouldn't pass plain text into a secure string. PowerShell returns this warning because the plain text password gets recorded in various logs. Once you have a secure string created, you need to pass it to the `PSCredential()` method to create the credential object. In the following example, the variable `$password` contains the secure string `$cred` contains the credential object.

PowerShell

```
$password = ConvertTo-SecureString "MyPlainTextPassword" -AsPlainText -Force  
$Cred = New-Object System.Management.Automation.PSCredential ("username",  
$password)
```

Now that you know how to create credential objects, you can add credential parameters to your PowerShell functions.

Adding a Credential Parameter

Just like any other parameter, you start off by adding it in the `param` block of your function. It's recommended that you name the parameter `$Credential` because that's what existing PowerShell cmdlets use. The type of the parameter should be `[System.Management.Automation.PSCredential]`.

The following example shows the parameter block for a function called `Get-Something`. It has two parameters: `$Name` and `$Credential`.

PowerShell

```
function Get-Something {
    param(
        $Name,
        [System.Management.Automation.PSCredential]$Credential
    )
}
```

The code in this example is enough to have a working credential parameter, however there are a few things you can add to make it more robust.

- Add the `[ValidateNotNull()]` validation attribute to check that the value being passed to `Credential`. If the parameter value is null, this attribute prevents the function from executing with invalid credentials.
- Add `[System.Management.Automation.Credential()]`. This allows you to pass in a username as a string and have an interactive prompt for the password.
- Set a default value for the `$Credential` parameter to `[System.Management.Automation.PSCredential]::Empty`. Your function you might be passing this `$Credential` object to existing PowerShell cmdlets. Providing a null value to the cmdlet called inside your function causes an error. Providing an empty credential object avoids this error.

Tip

Some cmdlets that accept a credential parameter do not support `[System.Management.Automation.PSCredential]::Empty` as they should. See the [Dealing with Legacy Cmdlets](#) section for a workaround.

Using credential parameters

The following example demonstrates how to use credential parameters. This example shows a function called `Set-RemoteRegistryValue`, which is out of [The Pester Book](#). This function defines the credential parameter using the techniques described in the previous section. The function calls `Invoke-Command` using the `$Credential` variable created by the function. This allows you to change the user who's running `Invoke-Command`. Because the default value of `$Credential` is an empty credential, the function can run without providing credentials.

PowerShell

```

function Set-RemoteRegistryValue {
    param(
        $ComputerName,
        $Path,
        $Name,
        $Value,
        [ValidateNotNull()]
        [System.Management.Automation.PSCredential]
        [System.Management.Automation.Credential()]
        $Credential = [System.Management.Automation.PSCredential]::Empty
    )
    $null = Invoke-Command -ComputerName $ComputerName -ScriptBlock {
        Set-ItemProperty -Path $Using:Path -Name $Using:Name -Value
        $Using:Value
    } -Credential $Credential
}

```

The following sections show different methods of providing credentials to `Set-RemoteRegistryValue`.

Prompting for credentials

Using `Get-Credential` in parentheses `()` at run time causes the `Get-Credential` to run first. You are prompted for a username and password. You could use the `Credential` or `UserName` parameters of `Get-Credential` to pre-populate the username and domain. The following example uses a technique called splatting to pass parameters to the `Set-RemoteRegistryValue` function. For more information about splatting, check out the [about_Splatting](#) article.

PowerShell

```

$remoteKeyParams = @{
    ComputerName = $Env:COMPUTERNAME
    Path = 'HKLM:\SOFTWARE\Microsoft\WebManagement\Server'
    Name = 'EnableRemoteManagement'
    Value = '1'
}

Set-RemoteRegistryValue @remoteKeyParams -Credential (Get-Credential)

```

A screenshot of the Windows PowerShell ISE interface. The top menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. Two tabs are open: Untitled1.ps1* and Untitled2.ps1*. The Untitled2.ps1* tab contains the following PowerShell script:

```
1 $remoteKeyParams = @{
2     ComputerName = $Env:COMPUTERNAME
3     Path = 'HKLM:\SOFTWARE\Microsoft\WebManagement\Server'
4     Name = 'EnableRemoteManagement'
5     Value = '1'
6 }
7 Set-RemoteRegistryValue @remoteKeyParams -Credential (Get-Credential)
```

The bottom status bar shows "Completed" and "Ln 8 Col 70 | 100%".

Using `(Get-Credential)` seems cumbersome. Normally, when you use the `Credential` parameter with only a username, the cmdlet automatically prompts for the password. The `[System.Management.Automation.Credential()]` attribute enables this behavior.

A screenshot of a PowerShell window titled "PowerShell". The command prompt shows "PS C:\>". The script is as follows:

```
$remoteKeyParams = @{
    ComputerName = $Env:COMPUTERNAME
    Path = 'HKLM:\SOFTWARE\Microsoft\WebManagement\Server'
    Name = 'EnableRemoteManagement'
    Value = '1'
}

Set-RemoteRegistryValue @remoteKeyParams -Credential duffney
```

A screenshot of the Windows PowerShell ISE interface. The top menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. Four tabs are open: Untitled1.ps1*, Untitled2.ps1*, Untitled3.ps1*, and Untitled4.ps1*. The Untitled4.ps1* tab contains the following PowerShell script:

```
1 $remoteKeyParams = @{
2     ComputerName = $Env:COMPUTERNAME
3     Path = 'HKLM:\SOFTWARE\Microsoft\WebManagement\Server'
4     Name = 'EnableRemoteManagement'
5     Value = '1'
6 }
7 Set-RemoteRegistryValue @remoteKeyParams -Credential duffney
```

The bottom status bar shows "Completed" and "Ln 8 Col 61 | 100%".

ⓘ Note

To set the registry value shown, these examples assume you have the **Web Server** features of Windows installed. Run `Install-WindowsFeature Web-Server` and `Install-WindowsFeature web-mgmt-tools` if required.

Provide credentials in a variable

You can also populate a credential variable ahead of time and pass it to the **Credential** parameter of `Set-RemoteRegistryValue` function. Use this method with Continuous Integration / Continuous Deployment (CI/CD) tools such as Jenkins, TeamCity, and Octopus Deploy. For an example using Jenkins, check out Hodge's blog post [Automating with Jenkins and PowerShell on Windows - Part 2 ↗](#).

This example uses the .NET method to create the credential object and a secure string to pass in the password.

PowerShell

```
$password = ConvertTo-SecureString "P@ssw0rd" -AsPlainText -Force
$Cred = New-Object System.Management.Automation.PSCredential ("duffney",
$password)

$remoteKeyParams = @{
    ComputerName = $Env:COMPUTERNAME
    Path = 'HKLM:\SOFTWARE\Microsoft\WebManagement\Server'
    Name = 'EnableRemoteManagement'
    Value = '1'
}

Set-RemoteRegistryValue @remoteKeyParams -Credential $Cred
```

For this example, the secure string is created using a clear text password. All of the previously mentioned CI/CD have a secure method of providing that password at run time. When using those tools, replace the plain text password with the variable defined within the CI/CD tool you use.

Run without credentials

Since `$Credential` defaults to an empty credential object, you can run the command without credentials, as shown in this example:

PowerShell

```

$remoteKeyParams = @{
    ComputerName = $Env:COMPUTERNAME
    Path = 'HKLM:\SOFTWARE\Microsoft\WebManagement\Server'
    Name = 'EnableRemoteManagement'
    Value = '1'
}

Set-RemoteRegistryValue @remoteKeyParams

```

Dealing with legacy cmdlets

Not all cmdlets support credential objects or allow empty credentials. Instead, the cmdlet wants username and password parameters as strings. There are a few ways to work around this limitation.

Using if-else to handle empty credentials

In this scenario, the cmdlet you want to run doesn't accept an empty credential object. This example adds the **Credential** parameter to `Invoke-Command` only if it's not empty. Otherwise, it runs the `Invoke-Command` without the **Credential** parameter.

PowerShell

```

function Set-RemoteRegistryValue {
    param(
        $ComputerName,
        $Path,
        $Name,
        $Value,
        [ValidateNotNull()]
        [System.Management.Automation.PSCredential]
        [System.Management.Automation.Credential()]
        $Credential = [System.Management.Automation.PSCredential]::Empty
    )

    if($Credential -ne [System.Management.Automation.PSCredential]::Empty) {
        Invoke-Command -ComputerName:$ComputerName -Credential:$Credential
    }
        Set-ItemProperty -Path $Using:Path -Name $Using:Name -Value
$Using:Value
    }
} else {
    Invoke-Command -ComputerName:$ComputerName {
        Set-ItemProperty -Path $Using:Path -Name $Using:Name -Value
$Using:Value
    }
}

```

```
    }  
}
```

Using splatting to handle empty credentials

This example uses parameter splatting to call the legacy cmdlet. The `$Credential` object is conditionally added to the hash table for splatting and avoids the need to repeat the `Invoke-Command` script block. To learn more about splatting inside functions, see the [Splatting Parameters Inside Advanced Functions](#) blog post.

PowerShell

```
function Set-RemoteRegistryValue {  
    param(  
        $ComputerName,  
        $Path,  
        $Name,  
        $Value,  
        [ValidateNotNull()]  
        [System.Management.Automation.PSCredential]  
        [System.Management.Automation.Credential()]  
        $Credential = [System.Management.Automation.PSCredential]::Empty  
    )  
  
    $Splat = @{  
        ComputerName = $ComputerName  
    }  
  
    if ($Credential -ne  
        [System.Management.Automation.PSCredential]::Empty) {  
        $Splat['Credential'] = $Credential  
    }  
  
    $null = Invoke-Command -ScriptBlock {  
        Set-ItemProperty -Path $Using:Path -Name $Using:Name -Value  
        $Using:Value  
    } @splat  
}
```

Working with string passwords

The `Invoke-Sqlcmd` cmdlet is an example of a cmdlet that accepts a string as a password. `Invoke-Sqlcmd` allows you to run simple SQL insert, update, and delete statements. `Invoke-Sqlcmd` requires a clear-text username and password rather than a more secure credential object. This example shows how to extract the username and password from a credential object.

The `Get-AllSQLDatabases` function in this example calls the `Invoke-Sqlcmd` cmdlet to query a SQL server for all its databases. The function defines a **Credential** parameter with the same attribute used in the previous examples. Since the username and password exist within the `$Credential` variable, you can extract those values for use with `Invoke-Sqlcmd`.

The user name is available from the **UserName** property of the `$Credential` variable. To obtain the password, you have to use the `GetNetworkCredential()` method of the `$Credential` object. The values are extracted into variables that are added to a hash table used for splatting parameters to `Invoke-Sqlcmd`.

PowerShell

```
function Get-AllSQLDatabases {
    param(
        $SQLServer,
        [ValidateNotNull()]
        [System.Management.Automation.PSCredential]
        [System.Management.Automation.Credential()]
        $Credential = [System.Management.Automation.PSCredential]::Empty
    )

    $UserName = $Credential.UserName
    $Password = $Credential.GetNetworkCredential().Password

    $splat = @{
        UserName = $UserName
        Password = $Password
        ServerInstance = 'SQLServer'
        Query = "Select * from Sys.Databases"
    }

    Invoke-Sqlcmd @splat
}

$credSplat = @{
    TypeName = 'System.Management.Automation.PSCredential'
    ArgumentList = 'duffney',('P@ssw0rd' | ConvertTo-SecureString -AsPlainText -Force)
}
$Credential = New-Object @credSplat

Get-AllSQLDatabases -SQLServer SQL01 -Credential $Credential
```

Continued learning credential management

Creating and storing credential objects securely can be difficult. The following resources can help you maintain PowerShell credentials.

- [BetterCredentials ↗](#)
- [Azure Key Vault ↗](#)
- [Vault Project ↗](#)
- [SecretManagement module ↗](#)

Avoid assigning variables in expressions

Article • 11/17/2022

PowerShell allows you to use assignments within expressions by enclosing the assignment in parentheses `()`. PowerShell passes the assigned value through. For example:

```
PowerShell

# In an `if` conditional
if ($foo = Get-Item $PROFILE) { "$foo exists" }

# Property access
($profileFile = Get-Item $PROFILE).LastWriteTime

# You can even *assign* to such expressions.
($profileFile = Get-Item $PROFILE).LastWriteTime = Get-Date
```

ⓘ Note

While this syntax is allowed, its use is discouraged. There are cases where this does not work and the intent of the code author can be confusing to other code reviewers.

Limitations

The assignment case doesn't always work. When it doesn't work, the assignment is discarded. If you create an instance of a *mutable* value type and attempt to both save the instance in a variable and modify one of its properties in the same expression, the property assignment is discarded.

```
PowerShell

# create mutable value type
PS> Add-Type 'public struct Foo { public int x; }'

# Create an instance, store it in a variable, and try to modify its
# property.
# This assignment is effectively IGNORED.
PS> ($var = [Foo]::new()).x = 1
PS> $var.x
0
```

The difference is that you can't return a reference to the value. Essentially, `($var = [Foo]::new())` is equivalent to `($var = [Foo]::new(); $var)`. You're no longer performing a member access on the variable you're performing a member access on the variable's output, which is a copy.

The workaround is to create the instance and save it in a variable first, and then assign to the property via the variable:

PowerShell

```
# create mutable value type
PS> Add-Type 'public struct Foo { public int x; }'

# Create an instance and store it in a variable first
# and then modify its property via the variable.
PS> $var = [Foo]::new()
PS> $var.x = 1
PS> $var.x
1
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Avoid using Invoke-Expression

Article • 11/17/2022

The `Invoke-Expression` cmdlet should only be used as a *last resort*. In most scenarios, safer and more robust alternatives are available. Forums like Stack Overflow are filled with examples of `Invoke-Expression` misuse. Also note that **PSScriptAnalyzer** has a rule for this. For more information, see [AvoidUsingInvokeExpression](#).

Carefully consider the security implications. When a string from an untrusted source such as user input is passed directly to `Invoke-Expression`, arbitrary commands can be executed. Always consider a different, more robust and secure solution first.

Common scenarios

Consider the following usage scenarios:

- It's simpler to redirect PowerShell to execute something naturally. For example:

```
PowerShell  
Get-Content ./file.ps1 | Invoke-Expression
```

These cases are trivially avoidable. The script or code already exists in file or AST form, so you should write a script with parameters and invoke it directly instead of using `Invoke-Expression` on a string.

- Running a script from a trusted source. For example, running the install script from the PowerShell repository:

```
PowerShell  
Invoke-WebRequest https://aka.ms/install-powershell.ps1 | Invoke-Expression
```

You should only use this interactively. And, while this does make life simpler, this practice should be discouraged.

- Testing for parsing errors. The PowerShell team tests for parse errors in the source code using `Invoke-Expression` because that's the only way to turn a parse-time error into a runtime one.

Conclusion

Most other scripting languages have a way to evaluate a string as code, and as an interpreted language, PowerShell must have a way to dynamically run itself. But there's no good reason to use `Invoke-Expression` in a production environment.

References

- Stack Overflow discussion - [In what scenario was Invoke-Expression designed to be used? ↗](#)
- PowerShell Blog post - [Invoke-Expression Considered Harmful ↗](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

- [Open a documentation issue](#)
- [Provide product feedback](#)

Limitations of PowerShell transcripts

Article • 04/30/2025

Mixing `Write-Host` output with the output objects, strings, and PowerShell transcription is complicated. There is a subtle interaction between the script and how transcription works with PowerShell pipelines that can have unexpected results.

When you emit objects from your script the formatting of those objects is handled by `Out-Default`. But the formatting can occur after the script has completed and transcription has stopped. This means that the output doesn't get transcribed. Strings are handled differently. Sometimes string output is passed through formatting, but not always. `Write-Host` makes an immediate write to the host process. `Write-Output` is sent through the formatting system. Combining the output of complex objects with writes to the host makes it difficult to predict what gets logged in the transcript.

Scenario 1 - Output of a structured object after all other operations

Consider the following script and its output:

```
PowerShell

PS> Get-Content scenario1.ps1
Start-Transcript scenario1.log -UseMinimalHeader
Write-Host '1'
Write-Output '2'
Get-Location
Write-Host '4'
Write-Output '5'
Stop-Transcript

PS> ./scenario1.ps1
Transcript started, output file is scenario1.log
1
2

4
Path
-----
/Users/user1/src/projects/transcript
5
Transcript stopped, output file is
/Users/user1/src/projects/transcript/scenario1.log
```

The output to the console shows the output you expect, but not in the order you expect it.

`Write-Host 4` is visible before `Get-Location` because `Write-Host` is optimized to write directly to the host. There's code in transcription that copies the output to the transcript file and the console. Then we have the regular output of `Get-Location` and `Write-Output 5` sent as output of the script.

```
PowerShell

PS> Get-Content scenario1.log
*****
PowerShell transcript start
Start time: 20191106114858
*****
Transcript started, output file is s2
1
2

4
*****
PowerShell transcript end
End time: 20191106114858
*****
```

Since transcription is turned off before the script exits, it's not rendered in the transcript. The objects were sent to the next consumer in the pipeline. In this case, it's `Out-Default`, which PowerShell inserted automatically. To complicate things further, the output of strings is also optimized in the formatting system. The first `Write-Output 2` gets emitted and captured by the transcript. But the insertion of the `Get-Location` object causes its output to be pushed into the stack of things that need actual formatting, which sets a bit of state for any remaining objects that also may need formatting. This is why the second `Write-Output 5` doesn't get added to the transcript.

Scenario 2 - Move the object emission to the beginning

Consider the following script and its output:

```
PowerShell

PS> Get-Content scenario2.ps1
Start-Transcript scenario2.log -UseMinimalHeader
Get-Location
Write-Host '1'
Write-Output '2'
Get-Location
```

```
Write-Host '4'
Write-Output '5'
Stop-Transcript

PS> ./scenario2.ps1
Transcript started, output file is scenario2.log

1
4
Path
-----
/Users/user1/src/projects/transcript
2
5
Transcript stopped, output file is
/Users/user1/src/projects/transcript/scenario2.log
```

We can see that the `Write-Host` commands happen before anything, and then the objects start to come out. The `Write-Output` of a string forces the object to be rendered to the screen, but notice that the transcript contains only the output of `Write-Host`. That's because those string objects are piped to `Out-Default` for formatting after the script turned off transcription.

```
PowerShell

PS> Get-Content scenario2.log
*****
PowerShell transcript start
Start time: 20220606094609
*****
Transcript started, output file is s3

1
4
*****
PowerShell transcript end
End time: 20220606094609
*****
```

Scenario 3 - Object emitted at the end of the script

For this scenario, the output of the complex object is at the end of the script.

```
PowerShell

PS> Get-Content scenario3.ps1
Start-Transcript scenario3.log -UseMinimalHeader
Write-Host '1'
Write-Output '2'
Write-Host '4'
```

```
Write-Output '5'
Get-Location
Stop-Transcript

PS> ./scenario3.ps1
Transcript started, output file is scenario3.log
1
2
4
5

Path
-----
/Users/user1/src/projects/transcript
Transcript stopped, output file is
/Users/user1/src/projects/transcript/scenario3.log
```

The string output from both `Write-Host` and `Write-Output` makes it into the transcript. However, the output from `Get-Location` occurs after transcription has stopped.

```
*****
PowerShell transcript start
Start time: 20220606100342
*****
Transcript started, output file is scenario3.log
1
2
4
5

*****
PowerShell transcript end
End time: 20220606100342
*****
```

A way to ensure full transcription

This example is a slight variation on the original scenario, but now everything is logged to the transcript. The original code is wrapped in a script block and the formatter explicitly invoked via `Out-Default`.

```
PowerShell

PS> Get-Content scenario4.ps1
Start-Transcript scenario4.log -UseMinimalHeader
. {
    Write-Host '1'
```

```
Write-Output '2'
Get-Location
Write-Host '4'
Write-Output '5'
} | Out-Default
Stop-Transcript

PS> ./scenario4.ps1
Transcript started, output file is scenario4.log
1
2

4
Path
-----
/Users/user1/src/projects/transcript
5

Transcript stopped, output file is
/Users/user1/src/projects/transcript/scenario4.log
```

Notice that the last `Write-Host` call is still out of order, that's because of the optimization in `Write-Host` that doesn't go into the output stream.

```
PowerShell

PS> Get-Content scenario4.log
*****
PowerShell transcript start
Start time: 20220606101038
*****
Transcript started, output file is s5
1
2

4
Path
-----
/Users/user1/src/projects/transcript
5

*****
PowerShell transcript end
End time: 20220606101038
*****
```

Sample scripts for system administration

A collection of examples walks through scenarios for administering systems with PowerShell.

Working with objects



HOW-TO GUIDE

[Viewing object structure](#)

[Selecting parts of objects](#)

[Removing objects from the pipeline](#)

[Sorting objects](#)

[Creating .NET and COM objects](#)

[Using static classes and methods](#)

[Getting WMI objects with Get-CimInstance](#)

[Manipulating items directly](#)

Managing computers



HOW-TO GUIDE

[Changing computer state](#)

[Collecting information about computers](#)

[Creating Get-WinEvent queries with FilterHashtable](#)

Managing processes & services



HOW-TO GUIDE

[Managing processes with process cmdlets](#)

[Managing services](#)

[Working with printers](#)

[Performing networking tasks](#)

[Working with software installations](#)

[Decode a PowerShell command from a running process](#)

Working with output



[CONCEPT](#)

[Redirecting output](#)

[Using format commands to change output view](#)

Managing drives & files



[HOW-TO GUIDE](#)

[Managing current location](#)

[Managing PowerShell drives](#)

[Working with files and folders](#)

[Working with files folders and registry keys](#)

[Working with registry entries](#)

[Working with registry keys](#)

Creating UI elements



[HOW-TO GUIDE](#)

[Creating a custom input box](#)

[Creating a graphical date picker](#)

[Multiple selection list boxes](#)

[Selecting items from a list box](#)

Viewing object structure

Article • 12/09/2022

Because objects play such a central role in PowerShell, there are several native commands designed to work with arbitrary object types. The most important one is the `Get-Member` command.

The simplest technique for analyzing the objects that a command returns is to pipe the output of that command to the `Get-Member` cmdlet. The `Get-Member` cmdlet shows you the formal name of the object type and a complete listing of its members. The number of elements that are returned can sometimes be overwhelming. For example, a process object can have over 100 members.

The following command allows you to see all the members of a `Process` object and page through the output.

```
PowerShell  
  
Get-Process | Get-Member | Out-Host -Paging
```

```
Output  
  
TypeName: System.Diagnostics.Process  
  
Name                        MemberType       Definition  
----                        -----  
Handles                    AliasProperty  Handles = HandleCount  
Name                       AliasProperty  Name = ProcessName  
NPM                       AliasProperty  NPM = NonpagedSystemMemorySize  
PM                          AliasProperty  PM = PagedMemorySize  
VM                          AliasProperty  VM = VirtualMemorySize  
WS                          AliasProperty  WS = WorkingSet  
add_Disposed              Method           System.Void  
add_Disposed(Event...)  
...
```

We can make this long list of information more usable by filtering for elements we want to see. The `Get-Member` command lets you list only members that are properties. There are several forms of properties. The cmdlet displays properties of a type using the `MemberType` parameter with the value `Properties`. The resulting list is still very long, but a more manageable:

```
PowerShell
```

```
Get-Process | Get-Member -MemberType Properties
```

Output

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

Name	MemberType	Definition
Handles	AliasProperty	Handles = HandleCount
Name	AliasProperty	Name = ProcessName
...		
ExitCode	Property	System.Int32 ExitCode {get;}
...		
Handle	Property	System.IntPtr Handle {get;}
...		
CPU	ScriptProperty	System.Object CPU
{get=\$this.Total...}		
...		
Path	ScriptProperty	System.Object Path
{get=\$this.Main...}		
...		

ⓘ Note

The allowed values of MemberType are AliasProperty, CodeProperty, Property, NoteProperty, ScriptProperty, Properties, PropertySet, Method, CodeMethod, ScriptMethod, Methods, ParameterizedProperty, MemberSet, and All.

There are more than 60 properties for a process. By default, PowerShell determines how to display an object type using information stored in XML files that have names ending in `.format.ps1xml`. The formatting definition for process objects is stored in

`DotNetTypes.format.ps1xml`.

If you need to look at properties other than those that PowerShell displays by default, you can format the output using the `Format-*` cmdlets.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Selecting parts of objects

Article • 12/09/2022

You can use the `Select-Object` cmdlet to create new, custom PowerShell objects that contain properties selected from the objects you use to create them. Type the following command to create a new object that includes only the **Name** and **FreeSpace** properties of the **Win32_LogicalDisk** WMI class:

PowerShell

```
Get-CimInstance -Class Win32_LogicalDisk |  
    Select-Object -Property Name, FreeSpace
```

Output

Name	FreeSpace
---	-----
C:	50664845312

With `Select-Object` you can create calculated properties to display **FreeSpace** in gigabytes rather than bytes.

PowerShell

```
Get-CimInstance -Class Win32_LogicalDisk |  
    Select-Object -Property Name, @{  
        Label='FreeSpace'  
        Expression= {($_.FreeSpace/1GB).ToString('F2')}  
    }
```

Output

Name	FreeSpace
---	-----
C:	47.18

Removing objects from the pipeline

Article • 12/09/2022

In PowerShell, you often generate and pass along more objects to a pipeline than you want. You can specify the properties of particular objects to display using the `Format-*` cmdlets, but this doesn't help with the problem of removing entire objects from the display. You may want to filter objects before the end of a pipeline, so you can perform actions on only a subset of the initially generated objects.

PowerShell includes a `Where-Object` cmdlet that allows you to test each object in the pipeline and only pass it along the pipeline if it meets a particular test condition. Objects that don't pass the test are removed from the pipeline. You supply the test condition as the value of the `FilterScript` parameter.

Performing simple tests with `Where-Object`

The value of `FilterScript` is a *script block* - one or more PowerShell commands surrounded by braces (`{}`) - that evaluates to true or false. These script blocks can be simple, but creating them requires knowing about another PowerShell concept, comparison operators. A comparison operator compares the items that appear on each side of it. Comparison operators begin with a hyphen character (`-`) and are followed by a name. Basic comparison operators work on almost any kind of object. The more advanced comparison operators might only work on text or arrays.

ⓘ Note

By default, PowerShell comparison operators are case-insensitive.

Due to parsing considerations, symbols such as `<`, `>`, and `=` aren't used as comparison operators. Instead, comparison operators are comprised of letters. The basic comparison operators are listed in the following table.

[] Expand table

Comparison Operator	Meaning	Example (returns true)
<code>-eq</code>	is equal to	<code>1 -eq 1</code>
<code>-ne</code>	isn't equal to	<code>1 -ne 2</code>
<code>-lt</code>	Is less than	<code>1 -lt 2</code>

Comparison Operator	Meaning	Example (returns true)
-le	Is less than or equal to	1 -le 2
-gt	Is greater than	2 -gt 1
-ge	Is greater than or equal to	2 -ge 1
-like	Is like (wildcard comparison for text)	"file.doc" -like "f*.do?"
-notlike	isn't like (wildcard comparison for text)	"file.doc" -notlike "p*.doc"
-contains	Contains	1,2,3 -contains 1
-notcontains	doesn't contain	1,2,3 -notcontains 4

`Where-Object` script blocks use the special variable `$_` to refer to the current object in the pipeline. Here is an example of how it works. If you have a list of numbers, and only want to return the ones that are less than 3, you can use `Where-Object` to filter the numbers by typing:

```
1,2,3,4 | Where-Object {$_ -lt 3}
1
2
```

Filtering based on object properties

Since `$_` refers to the current pipeline object, we can access its properties for our tests.

As an example, we can look at the `Win32_SystemDriver` class in WMI. There might be hundreds of system drivers on a particular system, but you might only be interested in a particular set of the system drivers, such as those that are running. For the `Win32_SystemDriver` class the relevant property is `State`. You can filter the system drivers, selecting only the running ones by typing:

PowerShell

```
Get-CimInstance -Class Win32_SystemDriver |
    Where-Object {$_ .State -eq 'Running'}
```

This still produces a long list. You may want to filter to only select the drivers set to start automatically by testing the `StartMode` value as well:

PowerShell

```
Get-CimInstance -Class Win32_SystemDriver |  
    Where-Object {$_._State -eq "Running"} |  
    Where-Object {$_._StartMode -eq "Auto"}
```

Output

```
DisplayName : RAS Asynchronous Media Driver  
Name       : AsyncMac  
State      : Running  
Status     : OK  
Started    : True  
  
DisplayName : Audio Stub Driver  
Name       : audstub  
State      : Running  
Status     : OK  
Started    : True  
...
```

This gives us a lot of information we no longer need because we know that the drivers are running. In fact, the only information we probably need at this point are the name and the display name. The following command includes only those two properties, resulting in much simpler output:

PowerShell

```
Get-CimInstance -Class Win32_SystemDriver |  
    Where-Object {$_._State -eq "Running"} |  
    Where-Object {$_._StartMode -eq "Manual"} |  
    Format-Table -Property Name, DisplayName
```

Output

Name	DisplayName
AsyncMac	RAS Asynchronous Media Driver
bindflt	Windows Bind Filter Driver
bowser	Browser
CompositeBus	Composite Bus Enumerator Driver
condrv	Console Driver
HdAudAddService	Microsoft 1.1 UAA Function Driver for High Definition
Audio Service	
HDAudBus	Microsoft UAA Bus Driver for High Definition Audio
HidUsb	Microsoft HID Class Driver
HTTP	HTTP Service
igfx	igfx
IntcDAud	Intel(R) Display Audio

There are two `Where-Object` elements in the above command, but they can be expressed in a single `Where-Object` element using the `-and` logical operator, like this:

PowerShell

```
Get-CimInstance -Class Win32_SystemDriver |  
    Where-Object {($_.State -eq 'Running') -and ($_.StartMode -eq 'Manual')}  
|  
    Format-Table -Property Name, DisplayName
```

The standard logical operators are listed in the following table.

[+] Expand table

Logical Operator	Meaning	Example (returns true)
<code>-and</code>	Logical and; true if both sides are true	<code>(1 -eq 1) -and (2 -eq 2)</code>
<code>-or</code>	Logical or; true if either side is true	<code>(1 -eq 1) -or (1 -eq 2)</code>
<code>-not</code>	Logical not; reverses true and false	<code>-not (1 -eq 2)</code>
<code>!</code>	Logical not; reverses true and false	<code>!(1 -eq 2)</code>



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Sorting objects

Article • 12/09/2022

We can organize displayed data to make it easier to scan using the `Sort-Object` cmdlet. `Sort-Object` takes the name of one or more properties to sort on, and returns data sorted by the values of those properties.

Basic sorting

Consider the problem of listing subdirectories and files in the current directory. If we want to sort by `LastWriteTime` and then by `Name`, we can do it by typing:

PowerShell

```
Get-ChildItem |  
    Sort-Object -Property LastWriteTime, Name |  
    Format-Table -Property LastWriteTime, Name
```

Output

LastWriteTime	Name
11/6/2017 10:10:11 AM	.localization-config
11/6/2017 10:10:11 AM	.openpublishing.build.ps1
11/6/2017 10:10:11 AM	appveyor.yml
11/6/2017 10:10:11 AM	LICENSE
11/6/2017 10:10:11 AM	LICENSE-CODE
11/6/2017 10:10:11 AM	ThirdPartyNotices
11/6/2017 10:10:15 AM	tests
6/6/2018 7:58:59 PM	CONTRIBUTING.md
6/6/2018 7:58:59 PM	README.md
...	

You can also sort the objects in reverse order by specifying the `Descending` switch parameter.

PowerShell

```
Get-ChildItem |  
    Sort-Object -Property LastWriteTime, Name -Descending |  
    Format-Table -Property LastWriteTime, Name
```

Output

```
LastWriteTime          Name
-----
12/1/2018 10:13:50 PM  reference
12/1/2018 10:13:50 PM  dsc
...
6/6/2018 7:58:59 PM   README.md
6/6/2018 7:58:59 PM   CONTRIBUTING.md
11/6/2017 10:10:15 AM tests
11/6/2017 10:10:11 AM ThirdPartyNotices
11/6/2017 10:10:11 AM LICENSE-CODE
11/6/2017 10:10:11 AM LICENSE
11/6/2017 10:10:11 AM appveyor.yml
11/6/2017 10:10:11 AM .openpublishing.build.ps1
11/6/2017 10:10:11 AM .localization-config
```

Using hash tables

You can sort different properties in different orders using hash tables in an array. Each hash table uses an **Expression** key to specify the property name as string and an **Ascending** or **Descending** key to specify the sort order by `$true` or `$false`. The **Expression** key is mandatory. The **Ascending** or **Descending** key is optional.

The following example sorts objects in descending **LastWriteTime** order and ascending **Name** order.

PowerShell

```
Get-ChildItem |
  Sort-Object -Property @{
    Expression = 'LastWriteTime'; Descending = $true
  },
    @{
      Expression = 'Name'; Ascending = $true
    } |
  Format-Table -Property LastWriteTime, Name
```

Output

```
LastWriteTime          Name
-----
12/1/2018 10:13:50 PM  dsc
12/1/2018 10:13:50 PM  reference
11/29/2018 6:56:01 PM  .openpublishing.redirection.json
11/29/2018 6:56:01 PM  gallery
11/24/2018 10:33:22 AM developer
11/20/2018 7:22:19 PM  .markdownlint.json
...
```

You can also set a scriptblock to the **Expression** key. When running the `Sort-Object` cmdlet, the scriptblock is executed and the result is used for sorting.

The following example sorts objects in descending order by the time span between **CreationTime** and **LastWriteTime**.

PowerShell

```
Get-ChildItem |  
    Sort-Object -Property @{ Exp = { $_.LastWriteTime - $_.CreationTime };  
    Desc = $true } |  
    Format-Table -Property LastWriteTime, CreationTime
```

Output

LastWriteTime	CreationTime
12/1/2018 10:13:50 PM	11/6/2017 10:10:11 AM
12/1/2018 10:13:50 PM	11/6/2017 10:10:11 AM
11/7/2018 6:52:24 PM	11/6/2017 10:10:11 AM
11/7/2018 6:52:24 PM	11/6/2017 10:10:15 AM
11/3/2018 9:58:17 AM	11/6/2017 10:10:11 AM
10/26/2018 4:50:21 PM	11/6/2017 10:10:11 AM
11/17/2018 1:10:57 PM	11/29/2017 5:48:30 PM
11/12/2018 6:29:53 PM	12/7/2017 7:57:07 PM
...	

Tips

You can omit the **Property** parameter name as following:

PowerShell

```
Sort-Object LastWriteTime, Name
```

Besides, you can refer to `Sort-Object` by its built-in alias, `sort`:

PowerShell

```
sort LastWriteTime, Name
```

The keys in the hash tables for sorting can be abbreviated as following:

PowerShell

```
Sort-Object @{ e = 'LastWriteTime'; d = $true }, @{ e = 'Name'; a = $true }
```

In this example, the **e** stands for **Expression**, the **d** stands for **Descending**, and the **a** stands for **Ascending**.

To improve readability, you can place the hash tables into a separate variable:

PowerShell

```
$order = @(  
    @{ Expression = 'LastWriteTime'; Descending = $true }  
    @{ Expression = 'Name'; Ascending = $true }  
)  
  
Get-ChildItem |  
    Sort-Object $order |  
    Format-Table LastWriteTime, Name
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Creating .NET and COM objects

Article • 12/09/2022

This sample only runs on Windows platforms.

There are software components with .NET Framework and COM interfaces that enable you to perform many system administration tasks. PowerShell lets you use these components, so you aren't limited to the tasks that can be performed by using cmdlets. Many of the cmdlets in the initial release of PowerShell don't work against remote computers. We will demonstrate how to get around this limitation when managing event logs by using the .NET Framework `System.Diagnostics.EventLog` class directly from PowerShell.

Using New-Object for event log access

The .NET Framework Class Library includes a class named `System.Diagnostics.EventLog` that can be used to manage event logs. You can create a new instance of a .NET Framework class by using the `New-Object` cmdlet with the `TypeName` parameter. For example, the following command creates an event log reference:

PowerShell

```
New-Object -TypeName System.Diagnostics.EventLog
```

Output

Max(K)	Retain	OverflowAction	Entries	Name
-----	-----	-----	-----	-----

Although the command has created an instance of the `EventLog` class, the instance doesn't include any data. that's because we didn't specify a particular event log. How do you get a real event log?

Using constructors with New-Object

To refer to a specific event log, you need to specify the name of the log. `New-Object` has an `ArgumentList` parameter. The arguments you pass as values to this parameter are used by a special startup method of the object. The method is called a `constructor`

because it's used to construct the object. For example, to get a reference to the Application log, you specify the string 'Application' as an argument:

PowerShell

```
New-Object -TypeName System.Diagnostics.EventLog -ArgumentList Application
```

Output

Max(K)	Retain	OverflowAction	Entries	Name
-----	-----	-----	-----	-----
16,384	7	OverwriteOlder	2,160	Application

ⓘ Note

Since most of the .NET classes are contained in the **System** namespace, PowerShell automatically attempts to find classes you specify in the **System** namespace if it can't find a match for the typename you specify. This means that you can specify `Diagnostics.EventLog` instead of `System.Diagnostics.EventLog`.

Storing Objects in Variables

You might want to store a reference to an object, so you can use it in the current shell. Although PowerShell lets you do a lot of work with pipelines, lessening the need for variables, sometimes storing references to objects in variables makes it more convenient to manipulate those objects.

The output from any valid PowerShell command can be stored in a variable. Variable names always begin with `$`. If you want to store the Application log reference in a variable named `$AppLog`, type the name of the variable, followed by an equal sign and then type the command used to create the Application log object:

PowerShell

```
$AppLog = New-Object -TypeName System.Diagnostics.EventLog -ArgumentList Application
```

If you then type `$AppLog`, you can see that it contains the Application log:

PowerShell

```
$AppLog
```

Output

Max(K) Retain OverflowAction	Entries Name
-----	-----
16,384 7 OverwriteOlder	2,160 Application

Accessing a remote event log with New-Object

The commands used in the preceding section target the local computer; the `Get-EventLog` cmdlet can do that. To access the Application log on a remote computer, you must supply both the log name and a computer name (or IP address) as arguments.

PowerShell

```
$RemoteAppLog = New-Object -TypeName System.Diagnostics.EventLog  
Application, 192.168.1.81  
$RemoteAppLog
```

Output

Max(K) Retain OverflowAction	Entries Name
-----	-----
512 7 OverwriteOlder	262 Application

Now that we have a reference to an event log stored in the `$RemoteAppLog` variable, what tasks can we perform on it?

Clearing an event log with object methods

Objects often have methods that can be called to perform tasks. You can use `Get-Member` to display the methods associated with an object. The following command and selected output show some the methods of the `EventLog` class:

PowerShell

```
$RemoteAppLog | Get-Member -MemberType Method
```

Output

TypeName: System.Diagnostics.EventLog

Name	MemberType	Definition
---		-----
...		
Clear	Method	System.Void Clear()
Close	Method	System.Void Close()
...		
GetType	Method	System.Type GetType()
...		
ModifyOverflowPolicy	Method	System.Void
ModifyOverflowPolicy(Overfl...		
RegisterDisplayName	Method	System.Void RegisterDisplayName(String
...		
...		
ToString	Method	System.String ToString()
WriteEntry	Method	System.Void WriteEntry(String
message),...		
WriteEvent	Method	System.Void WriteEvent(EventInstance
in...		

The `Clear()` method can be used to clear the event log. When calling a method, you must always follow the method name by parentheses, even if the method doesn't require arguments. This lets PowerShell distinguish between the method and a potential property with the same name. Type the following to call the `Clear` method:

PowerShell

```
$RemoteAppLog.Clear()  
$RemoteAppLog
```

Output

Max(K)	Retain	OverflowAction	Entries	Name
-----	-----	-----	-----	-----
512	7	OverwriteOlder	0	Application

Notice that the event log was cleared and now has 0 entries instead of 262.

Creating COM objects with New-Object

You can use `New-Object` to work with Component Object Model (COM) components. Components range from the various libraries included with Windows Script Host (WSH) to ActiveX applications such as Internet Explorer that are installed on most systems.

`New-Object` uses .NET Framework Runtime-Callable Wrappers to create COM objects, so it has the same limitations that .NET Framework does when calling COM objects. To create a COM object, you need to specify the **ComObject** parameter with the Programmatic Identifier or **ProgId** of the COM class you want to use. A complete discussion of the limitations of COM use and determining what ProgIds are available on a system is beyond the scope of this user's guide, but most well-known objects from environments such as WSH can be used within PowerShell.

You can create the WSH objects by specifying these progids: **WScript.Shell**, **WScript.Network**, **Scripting.Dictionary**, and **Scripting.FileSystemObject**. The following commands create these objects:

PowerShell

```
New-Object -ComObject WScript.Shell  
New-Object -ComObject WScript.Network  
New-Object -ComObject Scripting.Dictionary  
New-Object -ComObject Scripting.FileSystemObject
```

Although most of the functionality of these classes is made available in other ways in Windows PowerShell, a few tasks such as shortcut creation are still easier to do using the WSH classes.

Creating a desktop shortcut with **WScript.Shell**

One task that can be performed quickly with a COM object is creating a shortcut. Suppose you want to create a shortcut on your desktop that links to the home folder for PowerShell. You first need to create a reference to **WScript.Shell**, which we will store in a variable named `$WshShell`:

PowerShell

```
$WshShell = New-Object -ComObject WScript.Shell
```

`Get-Member` works with COM objects, so you can explore the members of the object by typing:

PowerShell

```
$WshShell | Get-Member
```

Output

```
TypeName: System.__ComObject#{41904400-be18-11d3-a28b-00104bd35090}
```

Name	MemberType	Definition
AppActivate	Method	bool AppActivate (Variant,
Va...		
CreateShortcut	Method	IDispatch CreateShortcut
(str...		
...		

`Get-Member` has an optional `InputObject` parameter you can use instead of piping to provide input to `Get-Member`. You would get the same output as shown above if you instead used the command `Get-Member -InputObject $WshShell`. If you use `InputObject`, it treats its argument as a single item. This means that if you have several objects in a variable, `Get-Member` treats them as an array of objects. For example:

PowerShell

```
$a = 1,2,"three"  
Get-Member -InputObject $a
```

Output

TypeName: System.Object[]		
Name	MemberType	Definition
Count	AliasProperty	Count = Length
...		

The `WScript.Shell` `CreateShortcut` method accepts a single argument, the path to the shortcut file to create. We could type in the full path to the desktop, but there is an easier way. The desktop is normally represented by a folder named `Desktop` inside the home folder of the current user. Windows PowerShell has a variable `$HOME` that contains the path to this folder. We can specify the path to the home folder by using this variable, and then add the name of the `Desktop` folder and the name for the shortcut to create by typing:

PowerShell

```
$lnk = $WshShell.CreateShortcut("$HOME\Desktop\PSHome.lnk")
```

When you use something that looks like a variable name inside double-quotes, PowerShell tries to substitute a matching value. If you use single-quotes, PowerShell

doesn't try to substitute the variable value. For example, try typing the following commands:

PowerShell

```
"$HOME\Desktop\PSHome.lnk"
```

Output

```
C:\Documents and Settings\aka\Desktop\PSHome.lnk
```

PowerShell

```
'$HOME\Desktop\PSHome.lnk'
```

Output

```
$HOME\Desktop\PSHome.lnk
```

We now have a variable named `$lnk` that contains a new shortcut reference. If you want to see its members, you can pipe it to `Get-Member`. The output below shows the members we need to use to finish creating our shortcut:

PowerShell

```
$lnk | Get-Member
```

Output

```
TypeName: System.__ComObject#{f935dc23-1cf0-11d0-adb9-00c04fd58a0b}
Name          MemberType  Definition
----          -----      -----
...
Save          Method     void Save ()
...
TargetPath    Property   string TargetPath () {get} {set}
```

We need to specify the `TargetPath`, which is the application folder for PowerShell, and then save the shortcut by calling the `Save` method. The PowerShell application folder path is stored in the variable `$PSHOME`, so we can do this by typing:

PowerShell

```
$lnk.TargetPath = $PSHOME  
$lnk.Save()
```

Using Internet Explorer from PowerShell

Many applications, including the Microsoft Office family of applications and Internet Explorer, can be automated by using COM. The following examples illustrate some of the typical techniques and issues involved in working with COM-based applications.

You create an Internet Explorer instance by specifying the Internet Explorer ProgId, `InternetExplorer.Application`:

```
PowerShell  
  
$ie = New-Object -ComObject InternetExplorer.Application
```

This command starts Internet Explorer, but doesn't make it visible. If you type `Get-Process`, you can see that a process named `iexplore` is running. In fact, if you exit PowerShell, the process will continue to run. You must reboot the computer or use a tool like Task Manager to end the `iexplore` process.

① Note

COM objects that start as separate processes, commonly called *ActiveX executables*, may or may not display a user interface window when they start up. If they create a window but don't make it visible, like Internet Explorer, the focus usually moves to the Windows desktop. You must make the window visible to interact with it.

By typing `$ie | Get-Member`, you can view properties and methods for Internet Explorer. To see the Internet Explorer window, set the `Visible` property to `$true` by typing:

```
PowerShell  
  
$ie.Visible = $true
```

You can then navigate to a specific Web address using the `Navigate` method:

```
PowerShell  
  
$ie.Navigate("https://devblogs.microsoft.com/scripting/")
```

Using other members of the Internet Explorer object model, it's possible to retrieve text content from the Web page. The following command displays the HTML text in the body of the current Web page:

```
PowerShell  
  
$ie.Document.Body.InnerText
```

To close Internet Explorer from within PowerShell, call its `Quit()` method:

```
PowerShell  
  
$ie.Quit()
```

The `$ie` variable no longer contains a valid reference even though it still appears to be a COM object. If you attempt to use it, PowerShell returns an automation error:

```
PowerShell  
  
$ie | Get-Member
```

Output

```
Get-Member : Exception retrieving the string representation for property  
"Application" : "The object invoked has disconnected from its clients. (Exception  
from HRESULT: 0x80010108 (RPC_E_DISCONNECTED))"  
At line:1 char:16  
+ $ie | Get-Member <<<
```

You can either remove the remaining reference with a command like `$ie = $null`, or completely remove the variable by typing:

```
PowerShell  
  
Remove-Variable ie
```

ⓘ Note

There is no common standard for whether ActiveX executables exit or continue to run when you remove a reference to one. Depending on circumstances, such as whether the application is visible, whether an edited document is running in it, and

even whether PowerShell is still running, the application may or may not exit. For this reason, you should test termination behavior for each ActiveX executable you want to use in PowerShell.

Getting warnings about .NET Framework-wrapped COM objects

In some cases, a COM object might have an associated .NET Framework **Runtime-Callable Wrapper** (RCW) that's used by `New-Object`. Since the behavior of the RCW may be different from the behavior of the normal COM object, `New-Object` has a **Strict** parameter to warn you about RCW access. If you specify the **Strict** parameter and then create a COM object that uses an RCW, you get a warning message:

PowerShell

```
$xl = New-Object -ComObject Excel.Application -Strict
```

Output

```
New-Object : The object written to the pipeline is an instance of the type
"Microsoft.Office.Interop.Excel.ApplicationClass" from the component's primary
interop assembly. If
this type exposes different members than the IDispatch members , scripts
written to work with this
object might not work if the primary interop assembly isn't installed. At
line:1 char:17 + $xl =
New-Object <<< -ComObject Excel.Application -Strict
```

Although the object is still created, you are warned that it isn't a standard COM object.

Using static classes and methods

Article • 12/09/2022

Not all .NET Framework classes can be created using `New-Object`. For example, if you try to create a `System.Environment` or a `System.Math` object with `New-Object`, you will get the following error messages:

PowerShell

```
New-Object System.Environment
```

Output

```
New-Object : Constructor not found. Cannot find an appropriate constructor
for
type System.Environment.
At line:1 char:11
+ New-Object <<< System.Environment
```

PowerShell

```
New-Object System.Math
```

Output

```
New-Object : Constructor not found. Cannot find an appropriate constructor
for
type System.Math.
At line:1 char:11
+ New-Object <<< System.Math
```

These errors occur because there is no way to create a new object from these classes. These classes are reference libraries of methods and properties that don't change state. You don't need to create them, you simply use them. Classes and methods such as these are called *static classes* because they're not created, destroyed, or changed. To make this clear we will provide examples that use static classes.

Getting environment data with `System.Environment`

Usually, the first step in working with an object in Windows PowerShell is to use `Get-Member` to find out what members it contains. With static classes, the process is a little different because the actual class isn't an object.

Referring to the static `System.Environment` class

You can refer to a static class by surrounding the class name with square brackets. For example, you can refer to `System.Environment` by typing the name within brackets.

Doing so displays some generic type information:

PowerShell	Output								
[System.Environment]									
	<table><thead><tr><th>IsPublic</th><th>IsSerial</th><th>Name</th><th>BaseType</th></tr></thead><tbody><tr><td>True</td><td>False</td><td>Environment</td><td>System.Object</td></tr></tbody></table>	IsPublic	IsSerial	Name	BaseType	True	False	Environment	System.Object
IsPublic	IsSerial	Name	BaseType						
True	False	Environment	System.Object						

① Note

As we mentioned previously, Windows PowerShell automatically prepends '`System.`' to type names when you use `New-Object`. The same thing happens when using a bracketed type name, so you can specify `[System.Environment]` as `[Environment]`.

The `System.Environment` class contains general information about the working environment for the current process, which is `powershell.exe` when working within Windows PowerShell.

If you try to view details of this class by typing `[System.Environment] | Get-Member`, the object type is reported as being `System.RuntimeType`, not `System.Environment`:

PowerShell	Output
[System.Environment] <code>Get-Member</code>	
	TypeName: System.RuntimeType

To view static members with Get-Member, specify the **Static** parameter:

PowerShell

```
[System.Environment] | Get-Member -Static
```

Output

TypeName: System.Environment

Name	MemberType	Definition
-----	-----	-----
Equals	Method	static System.Boolean Equals(Object
ob...		
Exit	Method	static System.Void Exit(Int32
exitCode)		
...		
CommandLine	Property	static System.String CommandLine
{get;}		
CurrentDirectory	Property	static System.String CurrentDirectory
...		
ExitCode	Property	static System.Int32 ExitCode
{get;set;}		
HasShutdownStarted	Property	static System.Boolean
HasShutdownStart...		
MachineName	Property	static System.String MachineName
{get;}		
NewLine	Property	static System.String NewLine {get;}
OSVersion	Property	static System.OperatingSystem
OSVersio...		
ProcessorCount	Property	static System.Int32 ProcessorCount
{get;}		
StackTrace	Property	static System.String StackTrace {get;}
SystemDirectory	Property	static System.String SystemDirectory
{...		
TickCount	Property	static System.Int32 TickCount {get;}
UserDomainName	Property	static System.String UserDomainName
{g...		
UserInteractive	Property	static System.Boolean UserInteractive
...		
UserName	Property	static System.String UserName {get;}
Version	Property	static System.Version Version {get;}
WorkingSet	Property	static System.Int64 WorkingSet {get;}
TickCount		ExitCode

We can now select properties to view from System.Environment.

Displaying static properties of System.Environment

The properties of `System.Environment` are also static, and must be specified in a different way than normal properties. We use `::` to indicate to Windows PowerShell that we want to work with a static method or property. To see the command that was used to launch Windows PowerShell, we check the `CommandLine` property by typing:

PowerShell	[<code>System.Environment</code>]:: <code>CommandLine</code>
Output	"C:\Program Files\Windows PowerShell\v1.0\powershell.exe"

To check the operating system version, display the `OSVersion` property by typing:

PowerShell	[<code>System.Environment</code>]:: <code>OSVersion</code>												
Output	<table><thead><tr><th>Platform</th><th>ServicePack</th><th>Version</th><th>VersionString</th></tr></thead><tbody><tr><td>Win32NT</td><td>Service Pack 2</td><td>5.1.2600.131072</td><td>Microsoft</td></tr><tr><td>Windows...</td><td></td><td></td><td></td></tr></tbody></table>	Platform	ServicePack	Version	VersionString	Win32NT	Service Pack 2	5.1.2600.131072	Microsoft	Windows...			
Platform	ServicePack	Version	VersionString										
Win32NT	Service Pack 2	5.1.2600.131072	Microsoft										
Windows...													

We can check whether the computer is in the process of shutting down by displaying the `HasShutdownStarted` property:

PowerShell	[<code>System.Environment</code>]:: <code>HasShutdownStarted</code>
Output	False

Doing math with `System.Math`

The `System.Math` static class is useful for performing some mathematical operations. The class includes several useful methods, which we can display using `Get-Member`.

⚠ Note

System.Math has several methods with the same name, but they're distinguished by the type of their parameters.

Type the following command to list the methods of the **System.Math** class.

PowerShell

```
[System.Math] | Get-Member -Static -MemberType Methods
```

Output

TypeName: System.Math

Name	MemberType	Definition
Abs	Method	static System.Single Abs(Single value), static
Sy...		
Acos	Method	static System.Double Acos(Double d)
Asin	Method	static System.Double Asin(Double d)
Atan	Method	static System.Double Atan(Double d)
Atan2	Method	static System.Double Atan2(Double y, Double x)
BigMul	Method	static System.Int64 BigMul(Int32 a, Int32 b)
Ceiling	Method	static System.Double Ceiling(Double a), static
Sy...		
Cos	Method	static System.Double Cos(Double d)
Cosh	Method	static System.Double Cosh(Double value)
DivRem	Method	static System.Int32 DivRem(Int32 a, Int32 b,
Int3...		
Equals	Method	static System.Boolean Equals(Object objA, Object
...		
Exp	Method	static System.Double Exp(Double d)
Floor	Method	static System.Double Floor(Double d), static
Syst...		
IEEERemainder	Method	static System.Double IEEERemainder(Double x,
Doub...		
Log	Method	static System.Double Log(Double d), static
System...		
Log10	Method	static System.Double Log10(Double d)
Max	Method	static System.SByte Max(SByte val1, SByte val2),
...		
Min	Method	static System.SByte Min(SByte val1, SByte val2),
...		
Pow	Method	static System.Double Pow(Double x, Double y)
ReferenceEquals	Method	static System.Boolean ReferenceEquals(Object
objA...		
Round	Method	static System.Double Round(Double a), static
Syst...		
Sign	Method	static System.Int32 Sign(SByte value), static

```
Sys...
Sin           Method    static System.Double Sin(Double a)
Sinh          Method    static System.Double Sinh(Double value)
Sqrt          Method    static System.Double Sqrt(Double d)
Tan           Method    static System.Double Tan(Double a)
Tanh          Method    static System.Double Tanh(Double value)
Truncate      Method    static System.Decimal Truncate(Decimal d),
static...
```

This displays several mathematical methods. Here is a list of commands that demonstrate how some of the common methods work:

PowerShell

```
[System.Math]::Sqrt(9)
3
[System.Math]::Pow(2,3)
8
[System.Math]::Floor(3.3)
3
[System.Math]::Floor(-3.3)
-4
[System.Math]::Ceiling(3.3)
4
[System.Math]::Ceiling(-3.3)
-3
[System.Math]::Max(2,7)
7
[System.Math]::Min(2,7)
2
[System.Math]::Truncate(9.3)
9
[System.Math]::Truncate(-9.3)
-9
```

Getting WMI objects with Get-CimInstance

Article • 10/13/2023

This sample only applies to Windows platforms.

Windows Management Instrumentation (WMI) is a core technology for Windows system administration because it exposes a wide range of information in a uniform manner. Because of how much WMI makes possible, the PowerShell cmdlet for accessing WMI objects, `Get-CimInstance`, is one of the most useful for doing real work. We're going to discuss how to use the CIM cmdlets to access WMI objects and then how to use WMI objects to do specific things.

List WMI classes

The first problem most WMI users face is trying to find out what can be done with WMI. WMI classes describe the resources that can be managed. There are hundreds of WMI classes, some of which contain dozens of properties.

`Get-CimClass` addresses this problem by making WMI discoverable. You can get a list of the WMI classes available on the local computer by typing:

PowerShell

```
Get-CimClass -Namespace root/CIMV2 |  
    Where-Object CimClassName -Like Win32* |  
    Select-Object CimClassName
```

Output

```
CimClassName  
-----  
Win32_DeviceChangeEvent  
Win32_SystemConfigurationChangeEvent  
Win32_VolumeChangeEvent  
Win32_SystemTrace  
Win32_ProcessTrace  
Win32_ProcessStartTrace  
Win32_ProcessStopTrace  
Win32_ThreadTrace  
Win32_ThreadStartTrace
```

```
Win32_ThreadStopTrace
```

```
...
```

You can retrieve the same information from a remote computer using the **ComputerName** parameter, specifying a computer name or IP address:

```
PowerShell
```

```
Get-CimClass -Namespace root/CIMV2 -ComputerName 192.168.1.29
```

The class listing returned by remote computers may vary due to the specific operating system the computer is running and the particular WMI extensions are added by installed applications.

 **Note**

When using CIM cmdlets to connect to a remote computer, the remote computer must be running WMI and the account you are using must be in the local **Administrators** group on the remote computer. The remote system doesn't need to have PowerShell installed. This allows you to administer operating systems that aren't running PowerShell, but do have WMI available.

Displaying WMI class details

If you already know the name of a WMI class, you can use it to get information immediately. For example, one of the WMI classes commonly used for retrieving information about a computer is **Win32_OperatingSystem**.

```
PowerShell
```

```
Get-CimInstance -Class Win32_OperatingSystem
```

Output

SystemDirectory Version	Organization	BuildNumber	RegisteredUser	SerialNumber
C:\WINDOWS\system32 10.0.22621	Microsoft	22621	USER1	00330-80000- 0000-AA175

Although we're showing all of the parameters, the command can be expressed in a more succinct way. The **ComputerName** parameter isn't necessary when connecting to the local system. We show it to demonstrate the most general case and remind you about the parameter. The **Namespace** defaults to **root/CIMV2**, and can be omitted as well. Finally, most cmdlets allow you to omit the name of common parameters. With `Get-CimInstance`, if no name is specified for the first parameter, PowerShell treats it as the **Class** parameter. This means the last command could have been issued by typing:

PowerShell

```
Get-CimInstance Win32_OperatingSystem
```

The **Win32_OperatingSystem** class has many more properties than those displayed here. You can use `Get-Member` to see all the properties. The properties of a WMI class are automatically available like other object properties:

PowerShell

```
Get-CimInstance -Class Win32_OperatingSystem | Get-Member -MemberType Property
```

Output

Type	Name	MemberType	Definition
Property	BootDevice	string	BootDevice
Property	BuildNumber	string	BuildNumber
Property	BuildType	string	BuildType {get;}
Property	Caption	string	Caption {get;}
Property	CodeSet	string	CodeSet {get;}
Property	CountryCode	string	CountryCode {get;}
Property	CreationClassName	string	
Property	CreationClassName {get;}		
Property	CSCreationClassName	string	
Property	CSCreationClassName {get;}		
Property	CSDVersion	string	CSDVersion {get;}
Property	CSName	string	CSName {get;}
Property	CurrentTimeZone	int16	CurrentTimeZone {get;}
Property	DataExecutionPrevention_32BitApplications	bool	
Property	DataExecutionPrevention_32BitApplications {get;}		

```
DataExecutionPrevention_Available      Property   bool  
DataExecutionPrevention_Available {get;}  
...
```

Displaying non-default properties with Format cmdlets

If you want the information contained in the **Win32_OperatingSystem** class that isn't displayed by default, you can display it by using the **Format** cmdlets. For example, if you want to display available memory data, type:

PowerShell

```
Get-CimInstance -Class Win32_OperatingSystem | Format-Table -Property  
TotalVirtualMemorySize, TotalVisibleMemorySize, FreePhysicalMemory,  
FreeVirtualMemory, FreeSpaceInPagingFiles
```

Output

TotalVirtualMemorySize	TotalVisibleMemorySize	FreePhysicalMemory
41787920	16622096	9537952
-----	-----	-----
33071884	25056628	

ⓘ Note

Wildcards work with property names in **Format-Table**, so the final pipeline element can be reduced to **Format-Table -Property Total*Memory*, Free***

The memory data might be more readable if you format it as a list by typing:

PowerShell

```
Get-CimInstance -Class Win32_OperatingSystem | Format-List Total*Memory*,  
Free*
```

Output

TotalVirtualMemorySize :	41787920
TotalVisibleMemorySize :	16622096
FreePhysicalMemory :	9365296

```
FreeSpaceInPagingFiles : 25042952
FreeVirtualMemory      : 33013484
Name                  : Microsoft Windows 11
Pro|C:\Windows|\Device\Harddisk0\Partition2
```

Manipulating items directly

Article • 11/06/2024

The elements that you see in PowerShell drives, such as the files and folders or registry keys, are called *Items* in PowerShell. The cmdlets for working with them item have the noun **Item** in their names.

The output of the `Get-Command -Noun Item` command shows that there are nine PowerShell item cmdlets.

```
PowerShell
Get-Command -Noun Item

Output
CommandType      Name                           Definition
----           ----
Cmdlet          Clear-Item                   Clear-Item [-Path]
<String[]>...
Cmdlet          Copy-Item                   Copy-Item [-Path]
<String[]>...
Cmdlet          Get-Item                    Get-Item [-Path] <String[]>
...
Cmdlet          Invoke-Item                 Invoke-Item [-Path]
<String[]>...
Cmdlet          Move-Item                  Move-Item [-Path]
<String[]>...
Cmdlet          New-Item                   New-Item [-Path] <String[]>
...
Cmdlet          Remove-Item                Remove-Item [-Path]
<String[]>...
Cmdlet          Rename-Item                Rename-Item [-Path]
<String>...
Cmdlet          Set-Item                   Set-Item [-Path] <String[]>
...
```

Creating new items

To create a new item in the filesystem, use the `New-Item` cmdlet. Include the **Path** parameter with path to the item, and the **ItemType** parameter with a value of `File` or `Directory`.

For example, to create a new directory named `New.Directory` in the `C:\Temp` directory, type:

PowerShell

```
New-Item -Path C:\temp\New.Directory -ItemType Directory
```

Output

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

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d---	2006-05-18 11:29 AM		New.Directory

To create a file, change the value of the `ItemType` parameter to `File`. For example, to create a file named `file1.txt` in the `New.Directory` directory, type:

PowerShell

```
New-Item -Path C:\temp\New.Directory\file1.txt -ItemType File
```

Output

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

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a--	2006-05-18 11:44 AM	0	file1

You can use the same technique to create a new registry key. In fact, a registry key is easier to create because the only item type in the Windows registry is a key. (Registry entries are item *properties*.) For example, to create a key named `_Test` in the `CurrentVersion` subkey, type:

PowerShell

```
New-Item -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\_Test
```

Output

```
Hive:  
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
```

SKC	VC	Name	Property
-----	----	------	----------

```
---  ---  ---  
0  0 _Test
```

```
{}
```

When typing a registry path, be sure to include the colon (:) in the PowerShell drive names, `HKLM:` and `HKCU:`. Without the colon, PowerShell doesn't recognize the drive name in the path.

Why registry values aren't items

When you use the `Get-ChildItem` cmdlet to find the items in a registry key, you will never see actual registry entries or their values.

For example, the registry key

`HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run` usually contains several registry entries that represent applications that run when the system starts.

However, when you use `Get-ChildItem` to look for child items in the key, all you will see is the `OptionalComponents` subkey of the key:

PowerShell

```
Get-ChildItem HKLM:\Software\Microsoft\Windows\CurrentVersion\Run
```

Output

```
Hive:  
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\Microsoft\Wi  
ndows\CurrentVersion\Run  
SKC  VC Name          Property  
---  --  ---  
3    0  OptionalComponents  {}
```

Although it would be convenient to treat registry entries as items, you can't specify a path to a registry entry in a way that ensures that it's unique. The path notation doesn't distinguish between the registry subkey named **Run** and the **(Default)** registry entry in the **Run** subkey. Furthermore, because registry entry names can contain the backslash character (\), if registry entries were items, then you couldn't use the path notation to distinguish a registry entry named `Windows\CurrentVersion\Run` from the subkey that's located in that path.

Renaming existing items

To change the name of a file or folder, use the `Rename-Item` cmdlet. The following command changes the name of the `file1.txt` file to `fileOne.txt`.

PowerShell

```
Rename-Item -Path C:\temp\New.Directory\file1.txt fileOne.txt
```

The `Rename-Item` cmdlet can change the name of a file or a folder, but it can't move an item. The following command fails because it attempts to move the file from the `New.Directory` directory to the Temp directory.

PowerShell

```
Rename-Item -Path C:\temp\New.Directory\fileOne.txt C:\temp\fileOne.txt
```

Output

```
Rename-Item : can't rename because the target specified isn't a path.  
At line:1 char:12  
+ Rename-Item <<< -Path C:\temp\New.Directory\fileOne C:\temp\fileOne.txt
```

Moving items

To move a file or folder, use the `Move-Item` cmdlet.

For example, the following command moves the `New.Directory` directory from the `C:\temp` directory to the root of the `C:` drive. To verify that the item was moved, include the `PassThru` parameter of the `Move-Item` cmdlet. Without `PassThru`, the `Move-Item` cmdlet doesn't display any results.

PowerShell

```
Move-Item -Path C:\temp\New.Directory -Destination C:\ -PassThru
```

Output

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

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	2006-05-18 12:14 PM		New.Directory

Copying items

If you are familiar with the copy operations in other shells, you might find the behavior of the `Copy-Item` cmdlet in PowerShell to be unusual. When you copy an item from one location to another, `Copy-Item` doesn't copy its contents by default.

For example, if you copy the `New.Directory` directory from the C: drive to the `C:\temp` directory, the command succeeds, but the files in the `New.Directory` directory aren't copied.

PowerShell

```
Copy-Item -Path C:\New.Directory -Destination C:\temp
```

If you display the contents of `C:\temp\New.Directory`, you will find that it contains no files:

```
PS> Get-ChildItem -Path C:\temp\New.Directory  
PS>
```

Why doesn't the `Copy-Item` cmdlet copy the contents to the new location?

The `Copy-Item` cmdlet was designed to be generic; it isn't just for copying files and folders. Also, even when copying files and folders, you might want to copy only the container and not the items within it.

To copy all of the contents of a folder, include the `Recurse` parameter of the `Copy-Item` cmdlet in the command. If you have already copied the directory without its contents, add the `Force` parameter, which allows you to overwrite the empty folder.

PowerShell

```
Copy-Item -Path C:\New.Directory -Destination C:\temp -Recurse -Force -  
PassThru
```

Output

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

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	2006-05-18	1:53 PM	New.Directory

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

Mode	LastWriteTime	Length	Name
-	-----	-----	-----
-a--	2006-05-18 11:44 AM	0	file1

Deleting items

To delete files and folders, use the `Remove-Item` cmdlet. PowerShell cmdlets, such as `Remove-Item`, that can make significant, irreversible changes will often prompt for confirmation when you enter its commands. For example, if you try to remove the `New.Directory` folder, you will be prompted to confirm the command, because the folder contains files:

PowerShell

```
Remove-Item C:\temp\New.Directory
```

Output

Confirm

The item at C:\temp\New.Directory has children and the -Recurse parameter was not specified. If you continue, all children will be removed with the item. Are you sure you want to continue?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):

Because `Yes` is the default response, to delete the folder and its files, press the `Enter` key. To remove the folder without confirming, use the `Recurse` parameter.

PowerShell

```
Remove-Item C:\temp\New.Directory -Recurse
```

Executing items

PowerShell uses the `Invoke-Item` cmdlet to perform a default action for a file or folder. This default action is determined by the default application handler in the registry; the effect is the same as if you double-click the item in File Explorer.

For example, suppose you run the following command:

```
PowerShell
```

```
Invoke-Item C:\WINDOWS
```

An Explorer window that's located in `C:\Windows` appears, just as if you had double-clicked the `C:\Windows` folder.

If you invoke the `Boot.ini` file on a system prior to Windows Vista:

```
PowerShell
```

```
Invoke-Item C:\boot.ini
```

If the `.ini` file type is associated with Notepad, the `boot.ini` file opens in Notepad.

Changing computer state

Article • 12/18/2023

This sample only applies to Windows platforms.

To reset a computer in PowerShell, use either a standard command-line tool, WMI, or a CIM class. Although you are using PowerShell only to run the tool, learning how to change a computer's power state in PowerShell illustrates some of the important details about working with external tools in PowerShell.

Locking a computer

The only way to lock a computer directly with the standard available tools is to call the `LockWorkstation()` function in `user32.dll`:

```
PowerShell  
rundll32.exe user32.dll,LockWorkStation
```

This command immediately locks the workstation. It uses `rundll32.exe` to call the `LockWorkStation` function in `user32.dll`.

When you lock a workstation while Fast User Switching is enabled, such as on Windows XP, the computer displays the user logon screen rather than starting the current user's screensaver.

To shut down particular sessions on a Terminal Server, use the `tsshutdn.exe` command-line tool.

Logging off the current session

You can use several different techniques to log off of a session on the local system. The simplest way is to use the Remote Desktop/Terminal Services command-line tool, `logoff.exe` (For details, at the PowerShell prompt, type `logoff /?`). To log off the current active session, type `logoff` with no arguments.

You can also use the `shutdown.exe` tool with its `logoff` option:

```
PowerShell
```

```
shutdown.exe -1
```

Another option is to use WMI. The **Win32_OperatingSystem** class has a **Shutdown** method. Invoking the method with the 0 flag initiates logoff:

For more information, see the [Shutdown method](#) of the **Win32_OperatingSystem** class.

PowerShell

```
Get-CimInstance -ClassName Win32_OperatingSystem | Invoke-CimMethod -  
MethodName Shutdown
```

Shutting down or restarting a computer

Shutting down and restarting computers are similar tasks. Most command-line tools support both actions. Windows includes two command-line tools for restarting a computer. Use either `tsshutdn.exe` or `shutdown.exe` with appropriate arguments. You can get detailed usage information from `tsshutdn.exe /?` or `shutdown.exe /?`.

You can also perform shutdown and restart operations directly from PowerShell.

To shut down the computer, use the `Stop-Computer` command

PowerShell

```
Stop-Computer
```

To restart the operating system, use the `Restart-Computer` command

PowerShell

```
Restart-Computer
```

To force an immediate restart of the computer, use the `-Force` parameter.

PowerShell

```
Restart-Computer -Force
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Collecting information about computers

Article • 12/09/2022

This sample only applies to Windows platforms.

Cmdlets from **CimCmdlets** module are the most important cmdlets for general system management tasks. All critical subsystem settings are exposed through WMI. Furthermore, WMI treats data as objects that are in collections of one or more items. Because PowerShell also works with objects and has a pipeline that allows you to treat single or multiple objects in the same way, generic WMI access allows you to perform some advanced tasks with very little work.

Listing desktop settings

We'll begin with a command that collects information about the desktops on the local computer.

PowerShell

```
Get-CimInstance -ClassName Win32/Desktop
```

This returns information for all desktops, whether they're in use or not.

ⓘ Note

Information returned by some WMI classes can be very detailed, and often include metadata about the WMI class.

Because most of these metadata properties have names that begin with **Cim**, you can filter the properties using `Select-Object`. Specify the `-ExcludeProperty` parameter with "`Cim*`" as the value. For example:

PowerShell

```
Get-CimInstance -ClassName Win32/Desktop | Select-Object -ExcludeProperty "CIM*"
```

To filter out the metadata, use a pipeline operator (`|`) to send the results of the `Get-CimInstance` command to `Select-Object -ExcludeProperty "CIM*"`.

Listing BIOS Information

The WMI `Win32_BIOS` class returns fairly compact and complete information about the system BIOS on the local computer:

PowerShell

```
Get-CimInstance -ClassName Win32_BIOS
```

Listing Processor Information

You can retrieve general processor information by using WMI's `Win32_Processor` class, although you will likely want to filter the information:

PowerShell

```
Get-CimInstance -ClassName Win32_Processor | Select-Object -ExcludeProperty "CIM*"
```

For a generic description string of the processor family, you can just return the `SystemType` property:

PowerShell

```
Get-CimInstance -ClassName Win32_ComputerSystem | Select-Object -Property SystemType  
SystemType  
-----  
X86-based PC
```

Listing computer manufacturer and model

Computer model information is also available from `Win32_ComputerSystem`. The standard displayed output will not need any filtering to provide OEM data:

PowerShell

```
Get-CimInstance -ClassName Win32_ComputerSystem
```

Output

Name	PrimaryOwnerName	Domain	TotalPhysicalMemory	Model
Manufacturer				
MyPC	Jane Doe	WORKGROUP	804765696	DA243A-ABA 6415c1 NA910
Compaq	Presario	06		

Your output from commands such as this, which return information directly from some hardware, is only as good as the data you have. Some information isn't correctly configured by hardware manufacturers and may therefore be unavailable.

Listing installed hotfixes

You can list all installed hotfixes by using `Win32_QuickFixEngineering`:

PowerShell

```
Get-CimInstance -ClassName Win32_QuickFixEngineering
```

This class returns a list of hotfixes that looks like this:

Output

Source	Description	HotFixID	InstalledBy	InstalledOn	PSComputerName
Security Update	KB4048951	Administrator	12/16/2017	.	

For more succinct output, you may want to exclude some properties. Although you can use the `Get-CimInstance`'s **Property** parameter to choose only the **HotFixID**, doing so will actually return more information, because all the metadata is displayed by default:

PowerShell

```
Get-CimInstance -ClassName Win32_QuickFixEngineering -Property HotFixID
```

Output

InstalledOn	:
Caption	:
Description	:
InstallDate	:
Name	:
Status	:
CSName	:

```
FixComments      :  
HotFixID        : KB4533002  
InstalledBy     :  
ServicePackInEffect :  
PSCoputerName   :  
CimClass         : root/cimv2:Win32_QuickFixEngineering  
CimInstanceProperties : {Caption, Description, InstallDate, Name...}  
CimSystemProperties  :  
Microsoft.Management.Infrastructure.CimSystemProperties  
...
```

The additional data is returned, because the **Property** parameter in `Get-CimInstance` restricts the properties returned from WMI class instances, not the object returned to PowerShell. To reduce the output, use `Select-Object`:

PowerShell

```
Get-CimInstance -ClassName Win32_QuickFixEngineering -Property HotFixId |  
    Select-Object -Property HotFixId
```

Output

```
HotFixId  
-----  
KB4048951
```

Listing operating system version information

The **Win32_OperatingSystem** class properties include version and service pack information. You can explicitly select only these properties to get a version information summary from **Win32_OperatingSystem**:

PowerShell

```
Get-CimInstance -ClassName Win32_OperatingSystem |  
    Select-Object -Property  
BuildNumber, BuildType, OSType, ServicePackMajorVersion, ServicePackMinorVersion
```

You can also use wildcards with the **Property** parameter. Because all the properties beginning with either **Build** or **ServicePack** are important to use here, we can shorten this to the following form:

PowerShell

```
Get-CimInstance -ClassName Win32_OperatingSystem |  
    Select-Object -Property Build*,OSType,ServicePack*
```

Output

```
BuildNumber      : 18362  
BuildType       : Multiprocessor Free  
OSType          : 18  
ServicePackMajorVersion : 0  
ServicePackMinorVersion : 0
```

Listing local users and owner

General information about local users can be found with a selection of **Win32_OperatingSystem** class properties. You can explicitly select the properties to display like this:

PowerShell

```
Get-CimInstance -ClassName Win32_OperatingSystem |  
    Select-Object -Property NumberOfLicensedUsers, NumberOfUsers,  
    RegisteredUser
```

A more succinct version using wildcards is:

PowerShell

```
Get-CimInstance -ClassName Win32_OperatingSystem | Select-Object -Property  
*user*
```

Getting available disk space

To see the disk space and free space for local drives, you can use the **Win32_LogicalDisk** class. You need to see only instances with a **DriveType** of 3, the value WMI uses for fixed hard disks.

PowerShell

```
Get-CimInstance -ClassName Win32_LogicalDisk -Filter "DriveType=3"
```

Output

DeviceID	DriveType	ProviderName	VolumeName	Size	FreeSpace
PSComputerName					
C:	3		Local Disk	203912880128	65541357568 .
Q:	3		New Volume	122934034432	44298250240 .

PowerShell

```
Get-CimInstance -ClassName Win32_LogicalDisk -Filter "DriveType=3" |  
    Measure-Object -Property FreeSpace,Size -Sum |  
    Select-Object -Property Property,Sum
```

Output

Property	Sum
FreeSpace	109839607808
Size	326846914560

Getting logon session information

You can get general information about logon sessions associated with users through the **Win32_LogonSession** WMI class:

PowerShell

```
Get-CimInstance -ClassName Win32_LogonSession
```

Getting the user logged on to a computer

You can display the user logged on to a particular computer system using **Win32_ComputerSystem**. This command returns only the user logged on to the system desktop:

PowerShell

```
Get-CimInstance -ClassName Win32_ComputerSystem -Property UserName
```

Getting local time from a computer

You can retrieve the current local time on a specific computer using the **Win32_LocalTime** WMI class.

PowerShell

```
Get-CimInstance -ClassName Win32_LocalTime
```

Output

```
Day          : 23
DayOfWeek    : 1
Hour         : 8
Milliseconds : 
Minute        : 52
Month         : 12
Quarter       : 4
Second        : 55
WeekInMonth   : 4
Year          : 2019
PSComputerName :
```

Displaying service status

To view the status of all services on a specific computer, you can locally use the **Get-Service** cmdlet. For remote systems, you can use the **Win32_Service** WMI class. If you also use **Select-Object** to filter the results to **Status**, **Name**, and **DisplayName**, the output format is almost identical to that from **Get-Service**:

PowerShell

```
Get-CimInstance -ClassName Win32_Service |
    Select-Object -Property Status,Name,DisplayName
```

To allow the complete display of names for services with long names, use the **AutoSize** and **Wrap** parameters of **Format-Table**. These parameters optimize column width and allow long names to wrap instead of being truncated:

PowerShell

```
Get-CimInstance -ClassName Win32_Service |
    Format-Table -Property Status, Name, DisplayName -AutoSize -Wrap
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Creating Get-WinEvent queries with FilterHashtable

Article • 06/28/2023

This sample only applies to Windows platforms.

To read the original June 3, 2014 [Scripting Guy](#) blog post, see [Use FilterHashTable to Filter Event Log with PowerShell ↴](#).

This article is an excerpt of the original blog post and explains how to use the `Get-WinEvent` cmdlet's `FilterHashtable` parameter to filter event logs. PowerShell's `Get-WinEvent` cmdlet is a powerful method to filter Windows event and diagnostic logs. Performance improves when a `Get-WinEvent` query uses the `FilterHashtable` parameter.

When you work with large event logs, it's not efficient to send objects down the pipeline to a `Where-Object` command. Prior to PowerShell 6, the `Get-EventLog` cmdlet was another option to get log data. For example, the following commands are inefficient to filter the **Microsoft-Windows-Defrag** logs:

```
PowerShell

Get-EventLog -LogName Application | Where-Object Source -Match defrag

Get-WinEvent -LogName Application | Where-Object { $_.ProviderName -match
'defrag' }
```

The following command uses a hash table that improves the performance:

```
PowerShell

Get-WinEvent -FilterHashtable @{
    LogName='Application'
    ProviderName='*defrag'
}
```

Blog posts about enumeration

This article presents information about how to use enumerated values in a hash table. For more information about enumeration, read these [Scripting Guy](#) blog posts. To

create a function that returns the enumerated values, see [Enumerations and Values](#). For more information, see the [Scripting Guy series of blog posts about enumeration](#).

Hash table key-value pairs

To build efficient queries, use the `Get-WinEvent` cmdlet with the **FilterHashtable** parameter. **FilterHashtable** accepts a hash table as a filter to get specific information from Windows event logs. A hash table uses **key-value** pairs. For more information about hash tables, see [about_Hash_Tables](#).

If the **key-value** pairs are on the same line, they must be separated by a semicolon. If each **key-value** pair is on a separate line, the semicolon isn't needed. For example, this article places **key-value** pairs on separate lines and doesn't use semicolons.

This sample uses several of the **FilterHashtable** parameter's **key-value** pairs. The completed query includes **LogName**, **ProviderName**, **Keywords**, **Id**, and **Level**.

The accepted **key-value** pairs are shown in the following table and are included in the documentation for the [Get-WinEvent FilterHashtable](#) parameter.

The following table displays the key names, data types, and whether wildcard characters are accepted for a data value.

[] Expand table

Key name	Value data type	Accepts wildcard characters?
LogName	<code><String[]></code>	Yes
ProviderName	<code><String[]></code>	Yes
Path	<code><String[]></code>	No
Keywords	<code><Long[]></code>	No
ID	<code><Int32[]></code>	No
Level	<code><Int32[]></code>	No
StartTime	<code><DateTime></code>	No
EndTime	<code><DateTime></code>	No
UserID	<code><SID></code>	No
Data	<code><String[]></code>	No

Key name	Value data type	Accepts wildcard characters?
<named-data>	<String[]>	No

The <named-data> key represents a named event data field. For example, the Perflib event 1008 can contain the following event data:

XML

```
<EventData>
  <Data Name="Service">BITS</Data>
  <Data Name="Library">C:\Windows\System32\bitsperf.dll</Data>
  <Data Name="Win32Error">2</Data>
</EventData>
```

You can query for these events using the following command:

PowerShell

```
Get-WinEvent -FilterHashtable @{LogName='Application'; 'Service'='Bits'}
```

ⓘ Note

The ability to query for <named-data> was added in PowerShell 6.

Building a query with a hash table

To verify results and troubleshoot problems, it helps to build the hash table one **key-value** pair at a time. The query gets data from the **Application** log. The hash table is equivalent to `Get-WinEvent -LogName Application`.

To begin, create the `Get-WinEvent` query. Use the `FilterHashtable` parameter's **key-value** pair with the key, **LogName**, and the value, **Application**.

PowerShell

```
Get-WinEvent -FilterHashtable @{
    LogName='Application'
}
```

Continue to build the hash table with the **ProviderName** key. Usually, the **ProviderName** is the name that appears in the **Source** field in the **Windows Event Viewer**. For example,

.NET Runtime in the following screenshot:

Image of Windows Event Viewer sources

Update the hash table and include the key-value pair with the key, **ProviderName**, and the value, .NET Runtime.

PowerShell

```
Get-WinEvent -FilterHashtable @{
    LogName='Application'
    ProviderName='.NET Runtime'
}
```

ⓘ Note

For some event providers, the correct **ProviderName** can be obtained by looking on the **Details** tab in **Event Properties**. For example, events where the **Source** field shows **Defrag**, the correct **ProviderName** is **Microsoft-Windows-Defrag**.

If your query needs to get data from archived event logs, use the **Path** key. The **Path** value specifies the full path to the log file. For more information, see the **Scripting Guy** blog post, [Use PowerShell to Parse Saved Event Logs for Errors ↗](#).

Using enumerated values in a hash table

Keywords is the next key in the hash table. The **Keywords** data type is an array of the **[long]** value type that holds a large number. Use the following command to find the maximum value of **[long]**:

PowerShell

```
[long]::MaxValue
```

Output

```
9223372036854775807
```

For the **Keywords** key, PowerShell uses a number, not a string such as **Security**. **Windows Event Viewer** displays the **Keywords** as strings, but they're enumerated values. In the hash table, if you use the **Keywords** key with a string value, an error message is displayed.

Open the **Windows Event Viewer** and from the **Actions** pane, click on **Filter current log**.

The **Keywords** drop-down menu displays the available keywords, as shown in the following screenshot:

[Image of Windows Event Viewer keywords](#)

Use the following command to display the `StandardEventKeywords` property names.

```
PowerShell

[System.Diagnostics.Eventing.Reader.StandardEventKeywords] |
    Get-Member -Static -MemberType Property

Output

TypeName: System.Diagnostics.Eventing.Reader.StandardEventKeywords
Name           MemberType Definition
--           -- -- -
AuditFailure   Property  static
System.Diagnostics.Eventing.Reader.StandardEventKey...
AuditSuccess   Property  static
System.Diagnostics.Eventing.Reader.StandardEventKey...
CorrelationHint Property  static
System.Diagnostics.Eventing.Reader.StandardEventKey...
CorrelationHint2 Property static
System.Diagnostics.Eventing.Reader.StandardEventKey...
EventLogClassic Property static
System.Diagnostics.Eventing.Reader.StandardEventKey...
None          Property  static
System.Diagnostics.Eventing.Reader.StandardEventKey...
ResponseTime   Property  static
System.Diagnostics.Eventing.Reader.StandardEventKey...
Sqm            Property  static
System.Diagnostics.Eventing.Reader.StandardEventKey...
WdiContext     Property  static
System.Diagnostics.Eventing.Reader.StandardEventKey...
WdiDiagnostic  Property  static
System.Diagnostics.Eventing.Reader.StandardEventKey...
```

The enumerated values are documented in the [.NET Framework](#). For more information, see [StandardEventKeywords Enumeration](#).

The **Keywords** names and enumerated values are as follows:

[+] [Expand table](#)

Name	Value
AuditFailure	4503599627370496
AuditSuccess	9007199254740992
CorrelationHint2	18014398509481984
EventLogClassic	36028797018963968
Sqm	2251799813685248
WdiDiagnostic	1125899906842624
WdiContext	562949953421312
ResponseTime	281474976710656
None	0

Update the hash table and include the **key-value** pair with the key, **Keywords**, and the **EventLogClassic** enumeration value, **36028797018963968**.

PowerShell

```
Get-WinEvent -FilterHashtable @{
    LogName='Application'
    ProviderName='.NET Runtime'
    Keywords=36028797018963968
}
```

Keywords static property value (optional)

The **Keywords** key is enumerated, but you can use a static property name in the hash table query. Rather than using the returned string, the property name must be converted to a value with the **value__** property.

For example, the following script uses the **value__** property.

PowerShell

```
$C =
[System.Diagnostics.Eventing.Reader.StandardEventKeywords]::EventLogClassic
Get-WinEvent -FilterHashtable @{
    LogName='Application'
    ProviderName='.NET Runtime'
    Keywords=$C.value__}
```

Filtering by Event Id

To get more specific data, the query's results are filtered by **Event Id**. The **Event Id** is referenced in the hash table as the key **Id** and the value is a specific **Event Id**. The **Windows Event Viewer** displays the **Event Id**. This example uses **Event Id 1023**.

Update the hash table and include the **key-value** pair with the key, **Id** and the value, **1023**.

PowerShell

```
Get-WinEvent -FilterHashtable @{
    LogName='Application'
    ProviderName='.NET Runtime'
    Keywords=36028797018963968
    ID=1023
}
```

Filtering by Level

To further refine the results and include only events that are errors, use the **Level** key. **Windows Event Viewer** displays the **Level** as string values, but they're enumerated values. In the hash table, if you use the **Level** key with a string value, an error message is displayed.

Level has values such as **Error**, **Warning**, or **Informational**. Use the following command to display the **StandardEventLevel** property names.

PowerShell

```
[System.Diagnostics.Eventing.Reader.StandardEventLevel] |
    Get-Member -Static -MemberType Property
```

Output

```
Type Name: System.Diagnostics.Eventing.Reader.StandardEventLevel
```

Name	MemberType	Definition
Critical	Property	static System.Diagnostics.Eventing.Reader.StandardEventLevel Critical {get;}
Error	Property	static System.Diagnostics.Eventing.Reader.StandardEventLevel Error {get;}
Informational	Property	static System.Diagnostics.Eventing.Reader.StandardEventLevel Informational {get;}
LogAlways	Property	static

```
System.Diagnostics.Eventing.Reader.StandardEventLevel LogAlways {get;}  
Verbose      Property static  
System.Diagnostics.Eventing.Reader.StandardEventLevel Verbose {get;}  
Warning      Property static  
System.Diagnostics.Eventing.Reader.StandardEventLevel Warning {get;}
```

The enumerated values are documented in the [.NET Framework](#). For more information, see [StandardEventLevel Enumeration](#).

The **Level** key's names and enumerated values are as follows:

[+] Expand table

Name	Value
Verbose	5
Informational	4
Warning	3
Error	2
Critical	1
LogAlways	0

The hash table for the completed query includes the key, **Level**, and the value, **2**.

PowerShell

```
Get-WinEvent -FilterHashtable @{  
    LogName='Application'  
    ProviderName='.NET Runtime'  
    Keywords=36028797018963968  
    ID=1023  
    Level=2  
}
```

Level static property in enumeration (optional)

The **Level** key is enumerated, but you can use a static property name in the hash table query. Rather than using the returned string, the property name must be converted to a value with the **value__** property.

For example, the following script uses the **value__** property.

PowerShell

```
$C = [System.Diagnostics.Eventing.Reader.StandardEventLevel]::Informational
Get-WinEvent -FilterHashtable @{
    LogName='Application'
    ProviderName='.NET Runtime'
    Keywords=36028797018963968
    ID=1023
    Level=$C.value__
}
```

Managing processes with Process cmdlets

Article • 12/09/2022

This sample only applies to Windows PowerShell 5.1.

You can use the Process cmdlets in PowerShell to manage local and remote processes in PowerShell.

Getting processes

To get the processes running on the local computer, run a `Get-Process` with no parameters.

You can get particular processes by specifying their process names or process IDs. The following command gets the Idle process:

PowerShell

```
Get-Process -Id 0
```

Output

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

Although it's normal for cmdlets to return no data in some situations, when you specify a process by its `ProcessId`, `Get-Process` generates an error if it finds no matches, because the usual intent is to retrieve a known running process. If there is no process with that ID, it's likely that the ID is incorrect or that the process of interest has already exited:

PowerShell

```
Get-Process -Id 99
```

Output

```
Get-Process : No process with process ID 99 was found.  
At line:1 char:12  
+ Get-Process <<< -Id 99
```

You can use the `Name` parameter of the `Get-Process` cmdlet to specify a subset of processes based on the process name. The `Name` parameter can take multiple names in a comma-separated list and it supports the use of wildcards, so you can type name patterns.

For example, the following command gets process whose names begin with "ex."

PowerShell

```
Get-Process -Name ex*
```

Output

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
234	7	5572	12484	134	2.98	1684	EXCEL
555	15	34500	12384	134	105.25	728	explorer

Because the .NET `System.Diagnostics.Process` class is the foundation for PowerShell processes, it follows some of the conventions used by `System.Diagnostics.Process`. One of those conventions is that the process name for an executable never includes the `.exe` at the end of the executable name.

`Get-Process` also accepts multiple values for the `Name` parameter.

PowerShell

```
Get-Process -Name exp*,power*
```

Output

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
540	15	35172	48148	141	88.44	408	explorer
605	9	30668	29800	155	7.11	3052	powershell

You can use the `ComputerName` parameter of `Get-Process` to get processes on remote computers. For example, the following command gets the PowerShell processes on the local computer (represented by "localhost") and on two remote computers.

PowerShell

```
Get-Process -Name powershell -ComputerName localhost, Server01, Server02
```

Output

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
258	8	29772	38636	130		3700	powershell
398	24	75988	76800	572		5816	powershell
605	9	30668	29800	155	7.11	3052	powershell

The computer names aren't evident in this display, but they're stored in the **MachineName** property of the process objects that `Get-Process` returns. The following command uses the `Format-Table` cmdlet to display the process **Id**, **ProcessName** and **MachineName** (ComputerName) properties of the process objects.

PowerShell

```
Get-Process -Name powershell -ComputerName localhost, Server01, Server02 |  
Format-Table -Property Id, ProcessName, MachineName
```

Output

Id	ProcessName	MachineName
3700	powershell	Server01
3052	powershell	Server02
5816	powershell	localhost

This more complex command adds the **MachineName** property to the standard `Get-Process` display.

PowerShell

```
Get-Process powershell -ComputerName localhost, Server01, Server02 |  
Format-Table -Property Handles,  
@{Label="NPM(K)";Expression={[int]($_.NPM/1024)}},  
@{Label="PM(K)";Expression={[int]($_.PM/1024)}},  
@{Label="WS(K)";Expression={[int]($_.WS/1024)}},  
@{Label="VM(M)";Expression={[int]($_.VM/1MB)}},  
@{Label="CPU(s)";Expression={if ($_.CPU -ne $())  
{$_.CPU.ToString("N")}}},  
Id, ProcessName, MachineName -Auto
```

Output

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName	MachineName
258	8	29772	38636	130		3700	powershell	Server01
398	24	75988	76800	572		5816	powershell	localhost
605	9	30668	29800	155	7.11	3052	powershell	Server02

Stopping processes

PowerShell gives you flexibility for listing processes, but what about stopping a process?

The `Stop-Process` cmdlet takes a **Name** or **Id** to specify a process you want to stop. Your ability to stop processes depends on your permissions. Some processes can't be stopped. For example, if you try to stop the idle process, you get an error:

PowerShell

```
Stop-Process -Name Idle
```

Output

```
Stop-Process : Process 'Idle (0)' cannot be stopped due to the following
error:
Access is denied
At line:1 char:13
+ Stop-Process <<< -Name Idle
```

You can also force prompting with the **Confirm** parameter. This parameter is particularly useful if you use a wildcard when specifying the process name, because you may accidentally match some processes you don't want to stop:

PowerShell

```
Stop-Process -Name t*,e* -Confirm
```

Output

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "explorer (408)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):n
Confirm
Are you sure you want to perform this action?
```

```
Performing operation "Stop-Process" on Target "taskmgr (4072)".  
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"):n
```

Complex process manipulation is possible by using some of the object filtering cmdlets. Because a Process object has a **Responding** property that's true when it's no longer responding, you can stop all nonresponsive applications with the following command:

PowerShell

```
Get-Process | Where-Object -FilterScript {$_.Responding -eq $false} | Stop-  
Process
```

You can use the same approach in other situations. For example, suppose a secondary notification area application automatically runs when users start another application. You may find that this doesn't work correctly in Terminal Services sessions, but you still want to keep it in sessions that run on the physical computer console. Sessions connected to the physical computer desktop always have a session ID of 0, so you can stop all instances of the process that are in other sessions by using `Where-Object` and the process, `SessionId`:

PowerShell

```
Get-Process -Name BadApp | Where-Object -FilterScript {$_.SessionId -neq 0}  
| Stop-Process
```

The `Stop-Process` cmdlet doesn't have a `ComputerName` parameter. Therefore, to run a stop process command on a remote computer, you need to use the `Invoke-Command` cmdlet. For example, to stop the PowerShell process on the Server01 remote computer, type:

PowerShell

```
Invoke-Command -ComputerName Server01 {Stop-Process PowerShell}
```

Stopping All Other PowerShell Sessions

It may occasionally be useful to be able to stop all running PowerShell sessions other than the current session. If a session is using too many resources or is inaccessible (it may be running remotely or in another desktop session), you may not be able to directly stop it. If you try to stop all running sessions, however, the current session may be terminated instead.

Each PowerShell session has an environment variable PID that contains the Id of the Windows PowerShell process. You can check the \$PID against the Id of each session and terminate only Windows PowerShell sessions that have a different Id. The following pipeline command does this and returns the list of terminated sessions (because of the use of the **PassThru** parameter):

PowerShell

```
Get-Process -Name powershell | Where-Object -FilterScript {$_._Id -ne $PID} | Stop-Process -PassThru
```

Output

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
334	9	23348	29136	143	1.03	388	powershell
304	9	23152	29040	143	1.03	632	powershell
302	9	20916	26804	143	1.03	1116	powershell
335	9	25656	31412	143	1.09	3452	powershell
303	9	23156	29044	143	1.05	3608	powershell
287	9	21044	26928	143	1.02	3672	powershell

Starting, debugging, and waiting for processes

PowerShell also comes with cmdlets to start (or restart), debug a process, and wait for a process to complete before running a command. For information about these cmdlets, see the cmdlet help topic for each cmdlet.

See also

- [Get-Process](#)
- [Stop-Process](#)
- [Start-Process](#)
- [Wait-Process](#)
- [Debug-Process](#)
- [Invoke-Command](#)

Managing services

Article • 12/09/2022

This sample only applies to Windows PowerShell 5.1.

There are eight core **Service** cmdlets, designed for a wide range of service tasks . This article only looks at listing and changing running state for services. You can get a list of service cmdlets using `Get-Command *-Service`. You can find information about each cmdlet by using `Get-Help <Cmdlet-Name>`, such as `Get-Help New-Service`.

Getting services

You can get the services on a local or remote computer by using the `Get-Service` cmdlet. As with `Get-Process`, using the `Get-Service` command without parameters returns all services. You can filter by name, even using an asterisk as a wildcard:

```
PowerShell

PS> Get-Service -Name se*

Status      Name            DisplayName
----      ----            -----
Running    seclogon        Secondary Logon
Running    SENS            System Event Notification
Stopped   ServiceLayer    ServiceLayer
```

Because it isn't always apparent what the real name for the service is, you may find you need to find services by display name. You can search by specific name, use wildcards, or provide a list of display names:

```
PowerShell

PS> Get-Service -DisplayName se*

Status      Name            DisplayName
----      ----            -----
Running    lanmanserver    Server
Running    SamSs           Security Accounts Manager
Running    seclogon        Secondary Logon
Stopped   ServiceLayer    ServiceLayer
Running    wscsvc          Security Center

PS> Get-Service -DisplayName ServiceLayer, Server
```

Status	Name	DisplayName
Running	lanmanserver	Server
Stopped	ServiceLayer	ServiceLayer

Getting remote services

With Windows PowerShell, you can use the `ComputerName` parameter of the `Get-Service` cmdlet to get the services on remote computers. The `ComputerName` parameter accepts multiple values and wildcard characters, so you can get the services on multiple computers with a single command. For example, the following command gets the services on the `Server01` remote computer.

```
PowerShell
```

```
Get-Service -ComputerName Server01
```

Starting in PowerShell 6.0, the `*-Service` cmdlets don't have the `ComputerName` parameter. You can still get services on remote computers with PowerShell remoting. For example, the following command gets the services on the `Server02` remote computer.

```
PowerShell
```

```
Invoke-Command -ComputerName Server02 -ScriptBlock { Get-Service }
```

You can also manage services with the other `*-Service` cmdlets. For more information on PowerShell remoting, see [about_Remote](#).

Getting required and dependent services

The `Get-Service` cmdlet has two parameters that are very useful in service administration. The `DependentServices` parameter gets services that depend on the service.

The `RequiredServices` parameter gets services upon which the `LanmanWorkstation` service depends.

```
PowerShell
```

```
PS> Get-Service -Name LanmanWorkstation -RequiredServices
```

Status	Name	DisplayName
-----	-----	-----

Running	MRxSmb20	SMB 2.0 MiniRedirector
Running	bowser	Bowser
Running	MRxSmb10	SMB 1.x MiniRedirector
Running	NSI	Network Store Interface Service

The **DependentServices** parameter gets services that require the LanmanWorkstation service.

PowerShell

```
PS> Get-Service -Name LanmanWorkstation -DependentServices
```

Status	Name	DisplayName
Running	SessionEnv	Terminal Services Configuration
Running	Netlogon	Netlogon
Stopped	Browser	Computer Browser
Running	BITS	Background Intelligent Transfer Ser...

The following command gets all services that have dependencies. The **Format-Table** cmdlet to display the **Status**, **Name**, **RequiredServices**, and **DependentServices** properties of the services.

PowerShell

```
Get-Service -Name * | Where-Object {$_._RequiredServices -or
$_._DependentServices} |
Format-Table -Property Status, Name, RequiredServices, DependentServices -
Auto
```

Stopping, starting, suspending, and restarting services

The Service cmdlets all have the same general form. Services can be specified by common name or display name, and take lists and wildcards as values. To stop the print spooler, use:

PowerShell

```
Stop-Service -Name spooler
```

To start the print spooler after it's stopped, use:

PowerShell

```
Start-Service -Name spooler
```

To suspend the print spooler, use:

PowerShell

```
Suspend-Service -Name spooler
```

The `Restart-Service` cmdlet works in the same manner as the other Service cmdlets:

PowerShell

```
PS> Restart-Service -Name spooler
```

```
WARNING: Waiting for service 'Print Spooler (Spooler)' to finish starting...
WARNING: Waiting for service 'Print Spooler (Spooler)' to finish starting...
PS>
```

Notice that you get a repeated warning message about the Print Spooler starting up. When you perform a service operation that takes some time, PowerShell notifies you that it's still attempting to perform the task.

If you want to restart multiple services, you can get a list of services, filter them, and then perform the restart:

PowerShell

```
PS> Get-Service | Where-Object -FilterScript {$_._CanStop} | Restart-Service

WARNING: Waiting for service 'Computer Browser (Browser)' to finish
stopping...
WARNING: Waiting for service 'Computer Browser (Browser)' to finish
stopping...
Restart-Service : can't stop service 'Logical Disk Manager (dmserver)'
because
    it has dependent services. It can only be stopped if the Force flag is set.
At line:1 char:57
+ Get-Service | Where-Object -FilterScript {$_._CanStop} | Restart-Service
<<<
WARNING: Waiting for service 'Print Spooler (Spooler)' to finish starting...
WARNING: Waiting for service 'Print Spooler (Spooler)' to finish starting...
```

These Service cmdlets don't have a `ComputerName` parameter, but you can run them on a remote computer by using the `Invoke-Command` cmdlet. For example, the following command restarts the Spooler service on the Server01 remote computer.

PowerShell

```
Invoke-Command -ComputerName Server01 {Restart-Service Spooler}
```

Setting service properties

The `Set-Service` cmdlet changes the properties of a service on a local or remote computer. Because the service status is a property, you can use this cmdlet to start, stop, and suspend a service. The Set-Service cmdlet also has a `StartupType` parameter that lets you change the service startup type.

To use `Set-Service` on Windows Vista and later versions of Windows, open PowerShell with the **Run as administrator** option.

For more information, see [Set-Service](#)

See also

- [about_Remote](#)
- [Get-Service](#)
- [Set-Service](#)
- [Restart-Service](#)
- [Suspend-Service](#)

Working with printers in Windows

Article • 12/09/2022

This sample only applies to Windows platforms.

You can use PowerShell to manage printers using WMI and the `WScript.Network` COM object from WSH.

Listing printer connections

The simplest way to list the printers installed on a computer is to use the WMI `Win32_Printer` class:

PowerShell

```
Get-CimInstance -Class Win32_Printer
```

You can also list the printers using the `WScript.Network` COM object that's typically used in WSH scripts:

PowerShell

```
(New-Object -ComObject WScript.Network).EnumPrinterConnections()
```

Because this command returns a simple string collection of port names and printer device names without any distinguishing labels, it isn't easy to interpret.

Adding a network printer

To add a new network printer, use `WScript.Network`:

PowerShell

```
(New-Object -ComObject  
WScript.Network).AddWindowsPrinterConnection("\\\Printserver01\Xerox5")
```

Setting a default printer

To use WMI to set the default printer, find the printer in the `Win32_Printer` collection and then invoke the `SetDefaultPrinter` method:

PowerShell

```
$printer = Get-CimInstance -Class Win32_Printer -Filter "Name='HP LaserJet 5Si'"  
Invoke-CimMethod -InputObject $printer -MethodName SetDefaultPrinter
```

`WScript.Network` is a little simpler to use, because it has a `SetDefaultPrinter` method that takes only the printer name as an argument:

PowerShell

```
(New-Object -ComObject WScript.Network).SetDefaultPrinter('HP LaserJet 5Si')
```

Removing a printer connection

To remove a printer connection, use the `WScript.Network RemovePrinterConnection` method:

PowerShell

```
(New-Object -ComObject  
WScript.Network).RemovePrinterConnection("\Printserver01\Xerox5")
```



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

Performing networking tasks

Article • 02/12/2025

This sample only applies to Windows platforms.

Because TCP/IP is the most commonly used network protocol, most low-level network protocol administration tasks involve TCP/IP. In this section, we use PowerShell and WMI to do these tasks.

Listing IP addresses for a computer

To get all IP addresses in use on the local computer, use the following command:

PowerShell

```
Get-CimInstance -Class Win32_NetworkAdapterConfiguration -Filter  
IPEnabled=$true |  
Select-Object -ExpandProperty IPAddress
```

Since the **IPAddress** property of a **Win32_NetworkAdapterConfiguration** object is an array, you must use the **ExpandProperty** parameter of **Select-Object** to see the entire list of addresses.

Output

```
10.0.0.1  
fe80::60ea:29a7:a233:7cb7  
2601:600:a27f:a470:f532:6451:5630:ec8b  
2601:600:a27f:a470:e167:477d:6c5c:342d  
2601:600:a27f:a470:b021:7f0d:eab9:6299  
2601:600:a27f:a470:a40e:ebce:1a8c:a2f3  
2601:600:a27f:a470:613c:12a2:e0e0:bd89  
2601:600:a27f:a470:444f:17ec:b463:7edd  
2601:600:a27f:a470:10fd:7063:28e9:c9f3  
2601:600:a27f:a470:60ea:29a7:a233:7cb7  
2601:600:a27f:a470::2ec1
```

Using the **Get-Member** cmdlet, you can see that the **IPAddress** property is an array:

PowerShell

```
Get-CimInstance -Class Win32_NetworkAdapterConfiguration -Filter  
IPEnabled=$true |
```

```
Get-Member -Name IPAddress
```

Output

```
TypeName:  
Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_NetworkAdapterConfiguration  
  
Name      MemberType Definition  
----      -----  
IPAddress Property   string[] IPAddress {get;}
```

The **IPAddress** property for each network adapter is actually an array. The braces in the definition indicate that **IPAddress** isn't a **System.String** value, but an array of **System.String** values.

Listing IP configuration data

To display detailed IP configuration data for each network adapter, use the following command:

PowerShell

```
Get-CimInstance -Class Win32_NetworkAdapterConfiguration -Filter  
IPEnabled=$true
```

The default display for the network adapter configuration object is a very reduced set of the available information. For in-depth inspection and troubleshooting, use **Select-Object** or a formatting cmdlet, such as **Format-List**, to specify the properties to be displayed.

In modern TCP/IP networks you are probably not interested in IPX or WINS properties. You can use the **ExcludeProperty** parameter of **Select-Object** to hide properties with names that begin with "WINS" or "IPX".

PowerShell

```
Get-CimInstance -Class Win32_NetworkAdapterConfiguration -Filter  
IPEnabled=$true |  
Select-Object -ExcludeProperty IPX*,WINS*
```

This command returns detailed information about DHCP, DNS, routing, and other minor IP configuration properties.

Pinging computers

You can perform a simple ping against a computer by using `Win32_PingStatus`. The following command performs the ping, but returns lengthy output:

PowerShell

```
Get-CimInstance -Class Win32_PingStatus -Filter "Address='127.0.0.1'"
```

The response from `Win32_PingStatus` contains 29 properties. You can use `Format-Table` to select the properties that are most interesting to you. The `AutoSize` parameter of `Format-Table` resizes the table columns so that they display properly in PowerShell.

PowerShell

```
Get-CimInstance -Class Win32_PingStatus -Filter "Address='127.0.0.1'" | Format-Table -Property Address,ResponseTime,StatusCode -AutoSize
```

Output

Address	ResponseTime	StatusCode
127.0.0.1	0	0

A `StatusCode` of 0 indicates a successful ping.

You can use an array to ping multiple computers with a single command. Because there is more than one address, use the `ForEach-Object` to ping each address separately:

PowerShell

```
'127.0.0.1','localhost','bing.com' | ForEach-Object -Process { Get-CimInstance -Class Win32_PingStatus -Filter ("Address='$_'") | Select-Object -Property Address,ResponseTime,StatusCode }
```

You can use the same command format to ping all the addresses on a subnet, such as a private network that uses network number 192.168.1.0 and a standard Class C subnet mask (255.255.255.0). Only addresses in the range of 192.168.1.1 through 192.168.1.254 are legitimate local addresses (0 is always reserved for the network number and 255 is a subnet broadcast address).

To represent an array of the numbers from 1 through 254 in PowerShell, use the expression `1..254`. A complete subnet ping can be performed by adding each value in the range to a partial address in the ping statement:

```
PowerShell

1..254 | ForEach-Object -Process {
    Get-CimInstance -Class Win32_PingStatus -Filter ("Address='192.168.1.$_''")
} |
    Select-Object -Property Address,ResponseTime,StatusCode
```

Note that this technique for generating a range of addresses can be used elsewhere as well. You can generate a complete set of addresses in this way:

```
PowerShell

$ips = 1..254 | ForEach-Object -Process {'192.168.1.' + $_}
```

Retrieving network adapter properties

Earlier, we mentioned that you could retrieve general configuration properties using the **Win32_NetworkAdapterConfiguration** class. Although not strictly TCP/IP information, network adapter information such as MAC addresses and adapter types can be useful for understanding what's going on with a computer. To get a summary of this information, use the following command:

```
PowerShell

Get-CimInstance -Class Win32_NetworkAdapter -ComputerName .
```

Assigning the DNS domain for a network adapter

To assign the DNS domain for automatic name resolution, use the **SetDNSDomain** method of the **Win32_NetworkAdapterConfiguration**. The **Query** parameter of **Invoke-CimMethod** takes a WQL query string. The cmdlet calls the method specified on each instance returned by the query.

```
PowerShell
```

```
$wql = 'SELECT * FROM Win32_NetworkAdapterConfiguration WHERE  
IPEnabled=True'  
$args = @{ DnsDomain = 'fabrikam.com'}  
Invoke-CimMethod -MethodName SetDNSDomain -Arguments $args -Query $wql
```

Filtering on `IPEnabled=True` is necessary, because even on a network that uses only TCP/IP, several of the network adapter configurations on a computer aren't true TCP/IP adapters. They're general software elements supporting RAS, VPN, QoS, and other services for all adapters and thus don't have an address of their own.

Performing DHCP configuration tasks

Modifying DHCP details involves working with a set of network adapters, just as the DNS configuration does. There are several distinct actions you can perform using WMI.

Finding DHCP-enabled adapters

To find the DHCP-enabled adapters on a computer, use the following command:

PowerShell

```
Get-CimInstance -Class Win32_NetworkAdapterConfiguration -Filter  
"DHCPEnabled=$true"
```

To exclude adapters with IP configuration problems, you can retrieve only IP-enabled adapters:

PowerShell

```
Get-CimInstance -Class Win32_NetworkAdapterConfiguration -Filter  
"IPEnabled=$true and DHCPEnabled=$true"
```

Retrieving DHCP properties

Because DHCP-related properties for an adapter generally begin with `DHCP`, you can use the `Property` parameter of `Format-Table` to display only those properties:

PowerShell

```
Get-CimInstance -Class Win32_NetworkAdapterConfiguration -Filter  
"IPEnabled=$true and DHCPEnabled=$true" |
```

```
Format-Table -Property DHCP*
```

Enabling DHCP on each adapter

To enable DHCP on all adapters, use the following command:

PowerShell

```
$wql = 'SELECT * from Win32_NetworkAdapterConfiguration WHERE IPEnabled=True  
and DHCPEnabled=False'  
Invoke-CimMethod -MethodName ReleaseDHCPLease -Query $wql
```

Using the filter statement `IPEnabled=True and DHCPEnabled=False` avoids enabling DHCP where it's already enabled.

Releasing and renewing DHCP leases on specific adapters

Instances of the `Win32_NetworkAdapterConfiguration` class has `ReleaseDHCPLease` and `RenewDHCPLease` methods. Both are used in the same way. In general, use these methods if you only need to release or renew addresses for an adapter on a specific subnet. The easiest way to filter adapters on a subnet is to choose only the adapter configurations that use the gateway for that subnet. For example, the following command releases all DHCP leases on adapters on the local computer that are obtaining DHCP leases from 192.168.1.254:

PowerShell

```
$wql = 'SELECT * from Win32_NetworkAdapterConfiguration WHERE  
DHCPServer="192.168.1.1"'  
Invoke-CimMethod -MethodName ReleaseDHCPLease -Query $wql
```

The only change for renewing a DHCP lease is to use the `RenewDHCPLease` method instead of the `ReleaseDHCPLease` method:

PowerShell

```
$wql = 'SELECT * from Win32_NetworkAdapterConfiguration WHERE  
DHCPServer="192.168.1.1"'  
Invoke-CimMethod -MethodName RenewDHCPLease -Query $wql
```

ⓘ Note

When using these methods on a remote computer, be aware that you can lose access to the remote system if you are connected to it through the adapter with the released or renewed lease.

Releasing and renewing DHCP leases on all adapters

You can perform global DHCP address releases or renewals on all adapters by using the `Win32_NetworkAdapterConfiguration` methods, `ReleaseDHCPLeaseAll` and `RenewDHCPLeaseAll`. However, the command must apply to the WMI class, rather than a particular adapter, because releasing and renewing leases globally is performed on the class, not on a specific adapter. The `Invoke-CimMethod` cmdlet can call the methods of a class.

PowerShell

```
Invoke-CimMethod -ClassName Win32_NetworkAdapterConfiguration -MethodName ReleaseDHCPLeaseAll
```

You can use the same command format to invoke the `RenewDHCPLeaseAll` method:

PowerShell

```
Invoke-CimMethod -ClassName Win32_NetworkAdapterConfiguration -MethodName RenewDHCPLeaseAll
```

Creating a network share

To create a network share, use the `Create` method of `Win32_Share`:

PowerShell

```
Invoke-CimMethod -ClassName Win32_Share -MethodName Create -Arguments @{
    Path = 'C:\temp'
    Name = 'TempShare'
    Type = [uint32]0 #Disk Drive
    MaximumAllowed = [uint32]25
    Description = 'test share of the temp folder'
}
```

This is equivalent to the following `net share` command on Windows:

PowerShell

```
net share tempshare=C:\temp /users:25 /remark:"test share of the temp folder"
```

To call a method of a WMI class that takes parameters you must know what parameters are available and the types of those parameters. For example, you can list the methods of the **Win32_Class** with the following commands:

PowerShell

```
(Get-CimClass -ClassName Win32_Share).CimClassMethods
```

Output

Name	ReturnType	Parameters
Qualifiers		
-----	-----	-----

Create	UInt32	{Access, Description, MaximumAllowed, Name...}
	{Constructor, Implemented, MappingStrings, Stati...	
SetShareInfo	UInt32	{Access, Description, MaximumAllowed}
	{Implemented, MappingStrings}	
GetAccessMask	UInt32	{}
	{Implemented, MappingStrings}	
Delete	UInt32	{}
	{Destructor, Implemented, MappingStrings}	

Use the following command to list the parameters of the `Create` method.

PowerShell

```
(Get-CimClass -ClassName Win32_Share).CimClassMethods['Create'].Parameters
```

Output

Name	CimType	Qualifiers
ReferenceClassName		
-----	-----	-----

Access	Instance	{EmbeddedInstance, ID, In, MappingStrings...}
Description	String	{ID, In, MappingStrings, Optional}
MaximumAllowed	UInt32	{ID, In, MappingStrings, Optional}
Name	String	{ID, In, MappingStrings}
Password	String	{ID, In, MappingStrings, Optional}
Path	String	{ID, In, MappingStrings}
Type	UInt32	{ID, In, MappingStrings}

You can also read the documentation for [Create](#) method of the `Win32_Share` class.

Removing a network share

You can remove a network share with `Win32_Share`, but the process is slightly different from creating a share, because you need to retrieve the specific instance to be removed, rather than the `Win32_Share` class. The following example deletes the share `TempShare`:

PowerShell

```
$wql = 'SELECT * from Win32_Share WHERE Name="TempShare"'
Invoke-CimMethod -MethodName Delete -Query $wql
```

Connecting a Windows-accessible network drive

The `New-PSDrive` cmdlet can create a PowerShell drive that's mapped to a network share.

PowerShell

```
New-PSDrive -Name "X" -PSProvider "FileSystem" -Root "\\\Server01\Public"
```

However, drives created this way are only available to PowerShell session where they're created. To map a drive that's available outside of PowerShell (or to other PowerShell sessions), you must use the `Persist` parameter.

PowerShell

```
New-PSDrive -Persist -Name "X" -PSProvider "FileSystem" -Root
"\\\Server01\Public"
```

ⓘ Note

Persistently mapped drives may not be available when running in an elevated context. This is the default behavior of Windows UAC. For more information, see the following article:

- [Mapped drives aren't available from an elevated prompt when UAC is configured to Prompt for credentials](#)

Working with software installations

Article • 03/17/2023

Applications installed with the Windows Installer can be found through WMI's queries, but not all applications use the Windows Installer. The specific techniques for finding applications installed with other tools depends on the installer software.

For example, applications installed by copying the files to a folder on the computer usually can't be managed using techniques discussed here. You can manage these applications as files and folders using the techniques discussed in [Working With Files and Folders](#).

For software installed using an installer package, the Windows Installer can be found using the `Win32Reg_AddRemovePrograms` or the `Win32_Product` classes. However, both of these have problems. The `Win32Reg_AddRemovePrograms` is only available if you are using System Center Configuration Manager (SCCM). And the `Win32_Product` class can be slow and has side effects.

⊗ Caution

The `Win32_Product` class isn't query optimized. Queries that use wildcard filters cause WMI to use the MSI provider to enumerate all installed products then parse the full list sequentially to handle the filter. This also initiates a consistency check of packages installed, verifying and repairing the install. The validation is a slow process and may result in errors in the event logs. For more information seek [KB article 974524](#).

This article provides an alternative method for finding installed software.

Querying the Uninstall registry key to find installed software

Because most standard applications register an uninstaller with Windows, we can work with those locally by finding them in the Windows registry. There is no guaranteed way to find every application on a system. However, it's possible to find all programs with listings displayed in **Add or Remove Programs** in the following registry key:

`HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall`.

We can find the number of installed applications by counting the number of registry keys:

PowerShell

```
$UninstallPath = 'HKLM:\Software\Microsoft\Windows\CurrentVersion\Uninstall'  
(Get-ChildItem -Path $UninstallPath).Count
```

Output

```
459
```

We can search this list of applications further using a variety of techniques. To display the values of the registry values in the registry keys under `Uninstall`, use the `GetValue()` method of the registry keys. The value of the method is the name of the registry entry. For example, to find the display names of applications in the `Uninstall` key, use the following command:

PowerShell

```
Get-ChildItem -Path $UninstallPath |  
ForEach-Object -Process { $_.GetValue('DisplayName') } |  
Sort-Object
```

ⓘ Note

There is no guarantee that the `DisplayName` values are unique.

The following example produces output similar to the `Win32Reg_AddRemovePrograms` class:

PowerShell

```
Get-ChildItem $UninstallPath |  
ForEach-Object {  
    $ProdID = ($_.Name -split '\\')[-1]  
    Get-ItemProperty -Path "$UninstallPath\$ProdID" -ea SilentlyContinue  
    |  
        Select-Object -Property DisplayName, InstallDate, @{n='ProdID'; e={$ProdID}}, Publisher, DisplayVersion  
    } | Select-Object -First 3
```

For the sake of brevity, this example uses `Select-Object` to limit the number of items returned to three.

Output

```
DisplayName      : 7-Zip 22.01 (x64)
InstallDate     :
ProdID          : 7-Zip
Publisher        : Igor Pavlov
DisplayVersion   : 22.01

DisplayName      : AutoHotkey 1.1.33.10
InstallDate     :
ProdID          : AutoHotkey
Publisher        : Lexikos
DisplayVersion   : 1.1.33.10

DisplayName      : Beyond Compare 4.4.6
InstallDate     : 20230310
ProdID          : BeyondCompare4_is1
Publisher        : Scooter Software
DisplayVersion   : 4.4.6.27483
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Decode a PowerShell command from a running process

Article • 12/09/2022

This sample only runs on Windows platforms.

At times, you may have a PowerShell process running that's taking up a large amount of resources. This process could be running in the context of a [Task Scheduler](#) job or a [SQL Server Agent](#) job. Where there are multiple PowerShell processes running, it can be difficult to know which process represents the problem. This article shows how to decode a script block that a PowerShell process is currently running.

Create a long running process

To demonstrate this scenario, open a new PowerShell window and run the following code. It executes a PowerShell command that outputs a number every minute for 10 minutes.

```
PowerShell

powershell.exe -Command {
    $i = 1
    while ( $i -le 10 )
    {
        Write-Output -InputObject $i
        Start-Sleep -Seconds 60
        $i++
    }
}
```

View the process

The body of the command that PowerShell is executing is stored in the **CommandLine** property of the [Win32_Process](#) class. If the command is an encoded command, the **CommandLine** property contains the string "EncodedCommand". Using this information, the encoded command can be de-obfuscated via the following process.

Start PowerShell as Administrator. It's vital that PowerShell is running as administrator, otherwise no results are returned when querying the running processes.

Execute the following command to get all the PowerShell processes that have an encoded command:

```
PowerShell

$powerShellProcesses = Get-CimInstance -ClassName Win32_Process -Filter
'CommandLine LIKE "%EncodedCommand%"'
```

The following command creates a custom PowerShell object that contains the process ID and the encoded command.

```
PowerShell

$commandDetails = $powerShellProcesses | Select-Object -Property ProcessId,
@{
    Name      = 'EncodedCommand'
    Expression = {
        if ( $_.CommandLine -match 'encodedCommand (.*) -inputFormat' )
        {
            return $Matches[1]
        }
    }
}
```

Now the encoded command can be decoded. The following snippet iterates over the command details object, decodes the encoded command, and adds the decoded command back to the object for further investigation.

```
PowerShell

$commandDetails | ForEach-Object -Process {
    # Get the current process
    $currentProcess = $_

    # Convert the Base 64 string to a Byte Array
    $commandBytes =
[System.Convert]::FromBase64String($currentProcess.EncodedCommand)

    # Convert the Byte Array to a string
    $decodedCommand =
[System.Text.Encoding]::Unicode.GetString($commandBytes)

    # Add the decoded command back to the object
    $commandDetails |
        Where-Object -FilterScript { $_.ProcessId -eq
$currentProcess.ProcessId } |
            Add-Member -MemberType NoteProperty -Name DecodedCommand -Value
$decodedCommand
```

```
}
```

```
$commandDetails[0] | Format-List -Property *
```

The decoded command can now be reviewed by selecting the decoded command property.

Output

```
ProcessId      : 8752
EncodedCommand :
IAAKAAoACgAgAAoAIAAgACAAIAAkAGkAIAA9ACAAMQAgAAoACgAKACAAcGAgACAAIAAgAHcAaABp
AGwAZQAgACgAIAAkAGkAIAAtAG

wAZQAgADEAMAAgACKAIAAKAAoACgAgAAoAIAAgACAAIAB7ACAAcGAKAAoAIAAKACAAIAAgACAAIA
AgACAAIAAgACAAIABXAHIaQB0AGUALQBP

AHUAdABwAHUAdAAgAC0ASQBuAHAAdQB0AE8AYgBqAGUAYwB0ACAAJABpACAAcGAKAAoAIAAKACAA
IAAgACAAIAAgACAAIABTAHQAYQ

ByAHQALQBTAGwAZQB1AHAAIAAtAFMAZQBjAG8AbgBkAHMAIAA2ADAAIAAKAAoACgAgAAoAIAAgAC
AAIAAgACAAIAAgACQAaQArACsA
IAAKAAoACgAgAAoAIAAgACAAIAB9ACAAcGAKAAoAIAAKAA==

DecodedCommand :
    $i = 1
    while ( $i -le 10 )
    {
        Write-Output -InputObject $i
        Start-Sleep -Seconds 60
        $i++
    }
```

Redirecting output

Article • 05/14/2024

PowerShell provides several cmdlets that let you control data output directly. These cmdlets share two important characteristics.

First, they generally transform data to some form of text. They do this because they output the data to system components that require text input. This means they need to represent the objects as text. Therefore, the text is formatted as you see it in the PowerShell console window.

Second, these cmdlets use the PowerShell verb **Out** because they send information out from PowerShell to somewhere else.

Console output

By default, PowerShell sends data to the host window, which is exactly what the `Out-Host` cmdlet does. The primary use for the `Out-Host` cmdlet is paging. For example, the following command uses `Out-Host` to page the output of the `Get-Command` cmdlet:

```
PowerShell  
  
Get-Command | Out-Host -Paging
```

The host window display is outside of PowerShell. This is important because when data is sent out of PowerShell, it's actually removed. You can see this if you try to create a pipeline that pages data to the host window, and then attempt to format it as a list, as shown here:

```
PowerShell  
  
Get-Process | Out-Host -Paging | Format-List
```

You might expect the command to display pages of process information in list format. Instead, it displays the default tabular list:

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
101	5	1076	3316	32	0.05	2888	alg
...							

```
618      18    39348      51108   143    211.20    740 explorer
257      8     9752       16828    79     3.02     2560 explorer
...
<SPACE> next page; <CR> next line; Q quit
...
```

The `Out-Host` cmdlet sends the data directly to the console, so the `Format-List` command never receives anything to format.

The correct way to structure this command is to put the `Out-Host` cmdlet at the end of the pipeline as shown below. This causes the process data to be formatted in a list before being paged and displayed.

```
PowerShell
Get-Process | Format-List | Out-Host -Paging
```

Output

```
Id      : 2888
Handles : 101
CPU     : 0.046875
Name    : alg
...

Id      : 740
Handles : 612
CPU     : 211.703125
Name    : explorer

Id      : 2560
Handles : 257
CPU     : 3.015625
Name    : explorer
...
<SPACE> next page; <CR> next line; Q quit
...
```

This applies to all of the `Out` cmdlets. An `Out` cmdlet should always appear at the end of the pipeline.

ⓘ Note

All the `Out` cmdlets render output as text, using the formatting in effect for the console window, including line length limits.

Discarding output

The `Out-Null` cmdlet is designed to immediately discard any input it receives. This is useful for discarding unnecessary data that you get as a side-effect of running a command. When type the following command, you don't get anything back from the command:

```
PowerShell
```

```
Get-Command | Out-Null
```

The `Out-Null` cmdlet doesn't discard error output. For example, if you enter the following command, a message is displayed informing you that PowerShell doesn't recognize `Is-NotACommand`:

```
PS> Get-Command Is-NotACommand | Out-Null
Get-Command : 'Is-NotACommand' isn't recognized as a cmdlet, function,
operable program, or script file.
At line:1 char:12
+ Get-Command <<< Is-NotACommand | Out-Null
```

Printing data

`Out-Printer` is only available on Windows platforms.

You can print data using the `Out-Printer` cmdlet. The `Out-Printer` cmdlet uses your default printer if you don't provide a printer name. You can use any Windows-based printer by specifying its display name. There is no need for any kind of printer port mapping or even a real physical printer. For example, if you have the Microsoft Office document imaging tools installed, you can send the data to an image file by typing:

```
PowerShell
```

```
Get-Command -Name Get-* | Out-Printer -Name 'Microsoft Office Document Image
Writer'
```

Saving data

You can send output to a file instead of the console window using the `Out-File` cmdlet.

The following command line sends a list of processes to the file

```
C:\temp\processlist.txt:
```

```
PowerShell
```

```
Get-Process | Out-File -FilePath C:\temp\processlist.txt
```

The results of using the `Out-File` cmdlet may not be what you expect if you are used to traditional output redirection. To understand its behavior, you must be aware of the context in which the `Out-File` cmdlet operates.

On Window PowerShell 5.1, the `Out-File` cmdlet creates a Unicode file. Some tools, that expect ASCII files, don't work correctly with the default output format. You can change the default output format to ASCII using the `Encoding` parameter:

```
PowerShell
```

```
Get-Process | Out-File -FilePath C:\temp\processlist.txt -Encoding ascii
```

`Out-File` formats file contents to look like console output. This causes the output to be truncated just as it's in a console window in most circumstances. For example, if you run the following command:

```
PowerShell
```

```
Get-Command | Out-File -FilePath C:\temp\output.txt
```

The output will look like this:

```
Output
```

CommandType	Name	Definition
Cmdlet	Add-Content	Add-Content [-Path]
<String[...]		
Cmdlet	Add-History	Add-History [[-InputObject]
...		
...		

To get output that doesn't force line wraps to match the screen width, you can use the `Width` parameter to specify line width. Because `Width` is a 32-bit integer parameter, the maximum value it can have is 2147483647. Type the following to set the line width to this maximum value:

PowerShell

```
Get-Command | Out-File -FilePath C:\temp\output.txt -Width 2147483647
```

The `Out-File` cmdlet is most useful when you want to save output as it would have displayed on the console.

Using Format commands to change output view

Article • 12/09/2022

PowerShell has a set of cmdlets that allow you to control how properties are displayed for particular objects. The names of all the cmdlets begin with the verb `Format`. They let you select which properties you want to show.

PowerShell			
<code>Get-Command -Verb Format -Module Microsoft.PowerShell.Utility</code>			
Output			
CommandType	Name	Version	Source
-----	----	-----	-----
Cmdlet	Format-Custom	6.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Format-Hex	6.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Format-List	6.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Format-Table	6.1.0.0	Microsoft.PowerShell.Utility
Cmdlet	Format-Wide	6.1.0.0	Microsoft.PowerShell.Utility

This article describes the `Format-Wide`, `Format-List`, and `Format-Table` cmdlets.

Each object type in PowerShell has default properties that are used when you don't select the properties to display. Each cmdlet uses the same `Property` parameter to specify which properties you want displayed. Because `Format-Wide` only shows a single property, its `Property` parameter only takes a single value, but the `Property` parameter of `Format-List` and `Format-Table` accepts a list of property names.

In this example, the default output of `Get-Process` cmdlet shows that we've two instances of Internet Explorer running.

PowerShell			
<code>Get-Process -Name iexplore</code>			
The default format for <code>Process</code> objects displays the properties shown here:			

Output						
NPM(K)	PM(M)	WS(M)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	--	--	-----

32	25.52	10.25	13.11	12808	1	iexplore
52	11.46	26.46	3.55	21748	1	iexplore

Using Format-Wide for single-item output

The `Format-Wide` cmdlet, by default, displays only the default property of an object. The information associated with each object is displayed in a single column:

PowerShell

```
Get-Command -Verb Format | Format-Wide
```

Output

Format-Custom	Format-Hex
Format-List	Format-Table
Format-Wide	

You can also specify a non-default property:

PowerShell

```
Get-Command -Verb Format | Format-Wide -Property Noun
```

Output

Custom	Hex
List	Table
Wide	

Controlling Format-Wide display with column

With the `Format-Wide` cmdlet, you can only display a single property at a time. This makes it useful for displaying large lists in multiple columns.

PowerShell

```
Get-Command -Verb Format | Format-Wide -Property Noun -Column 3
```

Output

Custom
Table

Hex
Wide

List

Using Format-List for a list view

The `Format-List` cmdlet displays an object in the form of a listing, with each property labeled and displayed on a separate line:

PowerShell

```
Get-Process -Name iexplore | Format-List
```

Output

```
Id      : 12808
Handles : 578
CPU     : 13.140625
SI      : 1
Name    : iexplore

Id      : 21748
Handles : 641
CPU     : 3.59375
SI      : 1
Name    : iexplore
```

You can specify as many properties as you want:

PowerShell

```
Get-Process -Name iexplore | Format-List -Property
ProcessName,FileVersion,StartTime,Id
```

Output

```
ProcessName : iexplore
FileVersion : 11.00.18362.1 (WinBuild.160101.0800)
StartTime   : 10/22/2019 11:23:58 AM
Id         : 12808

ProcessName : iexplore
FileVersion : 11.00.18362.1 (WinBuild.160101.0800)
StartTime   : 10/22/2019 11:23:57 AM
Id         : 21748
```

Getting detailed information using Format-List with wildcards

The `Format-List` cmdlet lets you use a wildcard as the value of its `Property` parameter. This lets you display detailed information. Often, objects include more information than you need, which is why PowerShell doesn't show all property values by default. To show all properties of an object, use the `Format-List -Property *` command. The following command generates more than 60 lines of output for a single process:

PowerShell

```
Get-Process -Name iexplore | Format-List -Property *
```

Although the `Format-List` command is useful for showing detail, if you want an overview of output that includes many items, a simpler tabular view is often more useful.

Using Format-Table for tabular output

If you use the `Format-Table` cmdlet with no property names specified to format the output of the `Get-Process` command, you get exactly the same output as you do without a `Format` cmdlet. By default, PowerShell displays `Process` objects in a tabular format.

PowerShell

```
Get-Service -Name win* | Format-Table
```

Output

Status	Name	DisplayName
Running	WinDefend	Windows Defender Antivirus Service
Running	WinHttpAutoProxy	WinHTTP Web Proxy Auto-Discovery Se...
Running	Winmgmt	Windows Management Instrumentation
Running	WinRM	Windows Remote Management (WS-Manag...

Note

`Get-Service` is only available on Windows platforms.

Improving Format-Table output

Although a tabular view is useful for displaying lots of information, it may be difficult to interpret if the display is too narrow for the data. In the previous example, the output is truncated. If you specify the `AutoSize` parameter when you run the `Format-Table` command, PowerShell calculates column widths based on the actual data displayed. This makes the columns readable.

PowerShell

```
Get-Service -Name win* | Format-Table -AutoSize
```

Output

Status	Name	DisplayName
Running	WinDefend	Windows Defender Antivirus Service
Running	WinHttpAutoProxySvc	WinHTTP Web Proxy Auto-Discovery Service
Running	Winmgmt	Windows Management Instrumentation
Running	WinRM	Windows Remote Management (WS-Management)

The `Format-Table` cmdlet might still truncate data, but it only truncates at the end of the screen. Properties, other than the last one displayed, are given as much size as they need for their longest data element to display correctly.

PowerShell

```
Get-Service -Name win* |  
    Format-Table -Property Name, Status, StartType, DisplayName,  
    DependentServices -AutoSize
```

Output

Name	Status	StartType	DisplayName
ces	Running	Automatic	Windows Defender Antivirus Service
{} WinHttpAutoProxySvc	Running	Manual	WinHTTP Web Proxy Auto-Discovery Service {NcaSvc, iphl...
Winmgmt	Running	Automatic	Windows Management Instrumentation {vmmms, TPHKLO...

```
WinRM          Running Automatic Windows Remote Management (WS-
Management) {}
```

The `Format-Table` command assumes that properties are listed in order of importance. The cmdlet attempts to fully display the properties nearest the beginning. If the `Format-Table` command can't display all the properties, it removes some columns from the display. You can see this behavior in the `DependentServices` property previous example.

Wrapping Format-Table output in columns

You can force lengthy `Format-Table` data to wrap within its display column using the `Wrap` parameter. Using the `Wrap` parameter may not do what you expect, since it uses default settings if you don't also specify `AutoSize`:

PowerShell

```
Get-Service -Name win* |
    Format-Table -Property Name, Status, StartType, DisplayName,
    DependentServices -Wrap
```

Output

Name	Status	StartType	DisplayName
DependentServices			
ces			
---	-----	-----	-----

WinDefend	Running	Automatic	Windows Defender Antivirus Service
{}			
WinHttpAutoProxySvc	Running	Manual	WinHTTP Web Proxy Auto-Discovery
Service {NcaSvc,			
iphlpsvc}			
Winmgmt	Running	Automatic	Windows Management Instrumentation
{vmmms,			
TPHKLOAD,			
SUService,			
smstsmsgr...}			
WinRM	Running	Automatic	Windows Remote Management (WS-
Management) {}			

Using the `Wrap` parameter by itself doesn't slow down processing very much. However, using `AutoSize` to format a recursive file listing of a large directory structure can take a

long time and use lots of memory before displaying the first output items.

If you aren't concerned about system load, then **AutoSize** works well with the **Wrap** parameter. The initial columns still use as much width as needed to display items on one line, but the final column is wrapped, if necessary.

 **Note**

Some columns may not be displayed when you specify the widest columns first. For best results, specify the smallest data elements first.

In the following example, we specify the widest properties first.

PowerShell

```
Get-Process -Name iexplore |  
    Format-Table -Wrap -AutoSize -Property FileVersion, Path, Name, Id
```

Even with wrapping, the final **Id** column is omitted:

Output

FileVersion	Path
Nam	
e	
-----	----

11.00.18362.1 (WinBuild.160101.0800)	C:\Program Files (x86)\Internet
Explorer\IEXPLORE.EXE	iex

plo

re

11.00.18362.1 (WinBuild.160101.0800)	C:\Program Files\Internet
Explorer\iexplore.exe	iex

plo

re

Organizing table output

Another useful parameter for tabular output control is **GroupBy**. Longer tabular listings in particular may be hard to compare. The **GroupBy** parameter groups output based on

a property value. For example, we can group services by **StartType** for easier inspection, omitting the **StartType** value from the property listing:

PowerShell

```
Get-Service -Name win* | Sort-Object StartType | Format-Table -GroupBy StartType
```

Output

StartType: Automatic		
Status	Name	DisplayName
-----	---	-----
Running	WinDefend	Windows Defender Antivirus Service
Running	Winmgmt	Windows Management Instrumentation
Running	WinRM	Windows Remote Management (WS-Managem...

StartType: Manual		
Status	Name	DisplayName
-----	---	-----
Running	WinHttpAutoProxyS...	WinHTTP Web Proxy Auto-Discovery Serv...

Managing current location

Article • 12/18/2023

When navigating folder systems in File Explorer, you usually have a specific working location - namely, the current open folder. Items in the current folder can be manipulated easily by clicking them. For command-line interfaces such as Cmd.exe, when you are in the same folder as a particular file, you can access it by specifying a relatively short name, rather than needing to specify the entire path to the file. The current directory is called the working directory.

PowerShell uses the noun **Location** to refer to the working directory, and implements a family of cmdlets to examine and manipulate your location.

Getting your current location (Get-Location)

To determine the path of your current directory location, enter the `Get-Location` command:

```
PowerShell
Get-Location
Output
Path
-----
C:\Documents and Settings\PowerUser
```

ⓘ Note

The `Get-Location` cmdlet is similar to the `pwd` command in the BASH shell. The `Set-Location` cmdlet is similar to the `cd` command in Cmd.exe.

Setting your current location (Set-Location)

The `Get-Location` command is used with the `Set-Location` command. The `Set-Location` command allows you to specify your current directory location.

```
PowerShell
```

```
Set-Location -Path C:\Windows
```

After you enter the command, notice that you don't receive any direct feedback about the effect of the command. Most PowerShell commands that perform an action produce little or no output because the output isn't always useful. To verify that a successful directory change has occurred when you enter the `Set-Location` command, include the `PassThru` parameter when you enter the `Set-Location` command:

PowerShell

```
Set-Location -Path C:\Windows -PassThru
```

Output

Path

C:\WINDOWS

The `PassThru` parameter can be used with many Set commands in PowerShell to return information about the result for cases in which there is no default output.

You can specify paths relative to your current location in the same way as you would in most Unix and Windows command shells. In standard notation for relative paths, a period (.) represents your current folder, and a doubled period (...) represents the parent directory of your current location.

For example, if you are in the `C:\Windows` folder, a period (.) represents `C:\Windows` and double periods (...) represent `C:.` You can change from your current location to the root of the `C:` drive by typing:

PowerShell

```
Set-Location -Path .. -PassThru
```

Output

Path

C:\

The same technique works on PowerShell drives that aren't file system drives, such as `HKLM:.` You can set your location to the `HKLM\Software` key in the registry by typing:

```
PowerShell
```

```
Set-Location -Path HKLM:\SOFTWARE -PassThru
```

```
Output
```

```
Path
```

```
----
```

```
HKLM:\SOFTWARE
```

You can then change the directory location to the parent directory, using a relative path:

```
PowerShell
```

```
Set-Location -Path .. -PassThru
```

```
Output
```

```
Path
```

```
----
```

```
HKLM:\
```

You can type `Set-Location` or use any of the built-in PowerShell aliases for `Set-Location` (`cd`, `chdir`, `sl`). For example:

```
PowerShell
```

```
cd -Path C:\Windows
```

```
PowerShell
```

```
chdir -Path .. -PassThru
```

```
PowerShell
```

```
sl -Path HKLM:\SOFTWARE -PassThru
```

Saving and recalling recent locations (Push-Location and Pop-Location)

When changing locations, it's helpful to keep track of where you have been and to be able to return to your previous location. The `Push-Location` cmdlet in PowerShell creates an ordered history (a "stack") of directory paths where you have been, and you can step back through the history of directory paths using the `Pop-Location` cmdlet.

For example, PowerShell typically starts in the user's home directory.

```
PowerShell  
  
Get-Location  
  
Path  
----  
C:\Documents and Settings\PowerUser
```

① Note

The word **stack** has a special meaning in many programming settings, including .NET Framework. Like a physical stack of items, the last item you put onto the stack is the first item that you can pull off the stack. Adding an item to a stack is colloquially known as "pushing" the item onto the stack. Pulling an item off the stack is colloquially known as "popping" the item off the stack.

To push the current location onto the stack, and then move to the Local Settings folder, type:

```
PowerShell  
  
Push-Location -Path "Local Settings"
```

You can then push the Local Settings location onto the stack and move to the Temp folder by typing:

```
PowerShell  
  
Push-Location -Path Temp
```

You can verify that you changed directories by entering the `Get-Location` command:

```
PowerShell  
  
Get-Location
```

Output

Path

C:\Documents and Settings\PowerUser\Local Settings\Temp

You can then pop back into the most recently visited directory by entering the `Pop-Location` command, and verify the change by entering the `Get-Location` command:

PowerShell

`Pop-Location`

`Get-Location`

Output

Path

C:\Documents and Settings\me\Local Settings

Just as with the `Set-Location` cmdlet, you can include the `PassThru` parameter when you enter the `Pop-Location` cmdlet to display the directory that you entered:

PowerShell

`Pop-Location -PassThru`

Output

Path

C:\Documents and Settings\PowerUser

You can also use the Location cmdlets with network paths. If you have a server named FS01 with a share named Public, you can change your location by typing

PowerShell

`Set-Location \\FS01\Public`

or

PowerShell

```
Push-Location \\FS01\Public
```

You can use the `Push-Location` and `Set-Location` commands to change the location to any available drive. For example, if you have a local CD-ROM drive with drive letter D that contains a data CD, you can change the location to the CD drive by entering the `Set-Location D:` command.

If the drive is empty, you get the following error message:

PowerShell

```
Set-Location D:
```

Output

```
Set-Location : Cannot find path 'D:\' because it does not exist.
```

When you are using a command-line interface, it's not convenient to use File Explorer to examine the available physical drives. Also, File Explorer would not show you the all the PowerShell drives. PowerShell provides a set of commands for manipulating PowerShell drives.

Managing PowerShell drives

Article • 12/19/2023

This sample only applies to Windows platforms.

A *PowerShell drive* is a data store location that you can access like a filesystem drive in PowerShell. The PowerShell providers create some drives for you, such as the file system drives (including `C:` and `D:`), the registry drives (`HKCU:` and `HKLM:`), and the certificate drive (`Cert:`), and you can create your own PowerShell drives. These drives are useful, but they're available only within PowerShell. You can't access them using other Windows tools, such as File Explorer or `Cmd.exe`.

PowerShell uses the noun, **PSDrive**, for commands that work with PowerShell drives. For a list of the PowerShell drives in your PowerShell session, use the `Get-PSDrive` cmdlet.

```
PowerShell
Get-PSDrive

Output
Name      Provider      Root
CurrentLocation
----      -----      ---
A          FileSystem    A:\                                 . . .
Alias     Alias
C          FileSystem    C:\                               . . .
Settings\me
cert      Certificate   \
D          FileSystem    D:\
Env       Environment
Function   Function
HKCU      Registry      HKEY_CURRENT_USER
HKLM      Registry      HKEY_LOCAL_MACHINE
Variable   Variable
```

Although the drives in the display vary with the drives on your system, yours should look similar to the output of the `Get-PSDrive` command shown above.

filesystem drives are a subset of the PowerShell drives. You can identify the filesystem drives by the `FileSystem` entry in the `Provider` column. The filesystem drives in PowerShell are supported by the PowerShell `FileSystem` provider.

To see the syntax of the `Get-PSDrive` cmdlet, type a `Get-Command` command with the **Syntax** parameter:

PowerShell

```
Get-Command -Name Get-PSDrive -Syntax
```

Output

```
Get-PSDrive [[-Name] <String[]>] [-Scope <String>] [-PSProvider <String[]>]
[-V
erbose] [-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable <String>]
[-
OutVariable <String>] [-OutBuffer <Int32>]
```

The **PSProvider** parameter lets you display only the PowerShell drives that are supported by a particular provider. For example, to display only the PowerShell drives that are supported by the PowerShell FileSystem provider, type a `Get-PSDrive` command with the **PSProvider** parameter and the **FileSystem** value:

PowerShell

```
Get-PSDrive -PSProvider FileSystem
```

Output

Name	Provider	Root	CurrentLocation
A	FileSystem	A:\	
C	FileSystem	C:\	...nd
Settings\PowerUser			
D	FileSystem	D:\	

To view the PowerShell drives that represent registry hives, use the **PSProvider** parameter to display only the PowerShell drives that are supported by the PowerShell Registry provider:

PowerShell

```
Get-PSDrive -PSProvider Registry
```

Output

Name	Provider	Root	
CurrentLocation			-----
---	-----	-----	-----
HKCU	Registry	HKEY_CURRENT_USER	
HKLM	Registry	HKEY_LOCAL_MACHINE	

You can also use the standard Location cmdlets with the PowerShell drives:

PowerShell

```
Set-Location HKLM:\SOFTWARE
Push-Location .\Microsoft
Get-Location
```

Output

Path

```
-----
```

```
HKLM:\SOFTWARE\Microsoft
```

Adding new PowerShell drives

You can add your own PowerShell drives by using the `New-PSDrive` command. To get the syntax for the `New-PSDrive` command, enter the `Get-Command` command with the `Syntax` parameter:

PowerShell

```
Get-Command -Name New-PSDrive -Syntax
```

Output

```
New- [-Description <String>] [-Scope <String>] [-Credential <PSCredential>]
[-Verbose] [-Debug ]
[-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable
<String>]
[-OutBuffer <Int32>] [-WhatIf] [-Confirm]
```

To create a new PowerShell drive, you must supply three parameters:

- A name for the drive (you can use any valid PowerShell name)

- The PSProvider - use `FileSystem` for filesystem locations and `Registry` for registry locations
- The root, that is, the path to the root of the new drive

For example, you can create a drive named `Office` that's mapped to the folder that contains the Microsoft Office applications on your computer, such as `C:\Program Files\Microsoft Office\OFFICE11`. To create the drive, type the following command:

PowerShell

```
New-PSDrive -Name Office -PSProvider FileSystem -Root "C:\Program Files\Microsoft Office\OFFICE11"
```

Output

Name	Provider	Root
CurrentLocation		
Office	FileSystem	C:\Program Files\Microsoft Offic...

① Note

In general, paths aren't case-sensitive.

A PowerShell drive is accessed using its name followed by a colon (:).

A PowerShell drive can make many tasks much simpler. For example, some of the most important keys in the Windows registry have extremely long paths, making them cumbersome to access and difficult to remember. Critical configuration information resides under `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`. To view and change items in the CurrentVersion registry key, you can create a PowerShell drive that's rooted in that key by typing:

PowerShell

```
New-PSDrive -Name cvkey -PSProvider Registry -Root HKLM\Software\Microsoft\Windows\CurrentVersion
```

Output

Name	Provider	Root
CurrentLocation		

```
---- ----- ----  
---  
cvkey Registry HKLM\Software\Microsoft\Windows\...
```

You can then change location to the `cvkey:` drive as you would any other drive:

```
PowerShell
```

```
cd cvkey:
```

or:

```
PowerShell
```

```
Set-Location cvkey: -PassThru
```

```
Output
```

```
Path
```

```
----
```

```
cvkey:\
```

The `New-PSDrive` cmdlet adds the new drive only to the current PowerShell session. If you close the PowerShell window, the new drive is lost. To save a PowerShell drive, use the `Export-Console` cmdlet to export the current PowerShell session, and then use the `powershell.exe PSConsoleFile` parameter to import it. Or, add the new drive to your Windows PowerShell profile.

Deleting PowerShell drives

You can delete drives from PowerShell using the `Remove-PSDrive` cmdlet. For example, if you added the `Office:` PowerShell drive, as shown in the `New-PSDrive` topic, you can delete it by typing:

```
PowerShell
```

```
Remove-PSDrive -Name Office
```

To delete the `cvkey:` PowerShell drive, use the following command:

```
PowerShell
```

```
Remove-PSDrive -Name cvkey
```

However, you can't delete it while you are in the drive. For example:

PowerShell

```
cd office:  
Remove-PSDrive -Name Office
```

Output

```
Remove-PSDrive : Cannot remove drive 'Office' because it is in use.  
At line:1 char:15  
+ Remove-PSDrive <<< -Name Office
```

Adding and removing drives outside PowerShell

PowerShell detects filesystem drives that are added or removed in Windows, including:

- network drives that are mapped
- USB drives that are attached
- Drives that are deleted using the `net use` command or from a Windows Script Host (WSH) script

Working with files and folders

Article • 10/18/2023

Navigating through PowerShell drives and manipulating the items on them is similar to manipulating files and folders on Windows disk drives. This article discusses how to deal with specific file and folder manipulation tasks using PowerShell.

List all files and folders within a folder

You can get all items directly within a folder using `Get-ChildItem`. Add the optional **Force** parameter to display hidden or system items. For example, this command displays the direct contents of PowerShell Drive `c:`.

```
PowerShell
```

```
Get-ChildItem -Path C:\ -Force
```

The command lists only the directly contained items, much like using the `dir` command in `cmd.exe` or `ls` in a Unix shell. To show items in subfolder, you need to specify the **Recurse** parameter. The following command lists everything on the `c:` drive:

```
PowerShell
```

```
Get-ChildItem -Path C:\ -Force -Recurse
```

`Get-ChildItem` can filter items with its **Path**, **Filter**, **Include**, and **Exclude** parameters, but those are typically based only on name. You can perform complex filtering based on other properties of items using `Where-Object`.

The following command finds all executables within the Program Files folder that were last modified after October 1, 2005 and that are neither smaller than 1 megabyte nor larger than 10 megabytes:

```
PowerShell
```

```
Get-ChildItem -Path $Env:ProgramFiles -Recurse -Include *.exe |  
    Where-Object -FilterScript {  
        ($_.LastWriteTime -gt '2005-10-01') -and ($_.Length -ge 1mb) -and  
        ($_.Length -le 10mb)  
    }
```

Copying files and folders

Copying is done with `Copy-Item`. The following command backs up your PowerShell profile script:

```
PowerShell

if (Test-Path -Path $PROFILE) {
    Copy-Item -Path $PROFILE -Destination $($PROFILE -replace 'ps1$', 'bak')
}
```

The `Test-Path` command checks whether the profile script exists.

If the destination file already exists, the copy attempt fails. To overwrite a pre-existing destination, use the `Force` parameter:

```
PowerShell

if (Test-Path -Path $PROFILE) {
    Copy-Item -Path $PROFILE -Destination $($PROFILE -replace 'ps1$', 'bak') -
    Force
}
```

This command works even when the destination is read-only.

Folder copying works the same way. This command copies the folder `C:\temp\test1` to the new folder `C:\temp\DeleteMe` recursively:

```
PowerShell

Copy-Item C:\temp\test1 -Recurse C:\temp\DeleteMe
```

You can also copy a selection of items. The following command copies all `.txt` files contained anywhere in `C:\data` to `C:\temp\text`:

```
PowerShell

Copy-Item -Filter *.txt -Path C:\data -Recurse -Destination C:\temp\text
```

You can still run native commands like `xcopy.exe` and `robocopy.exe` to copy files.

Creating files and folders

Creating new items works the same on all PowerShell providers. If a PowerShell provider has more than one type of item—for example, the FileSystem PowerShell provider distinguishes between directories and files—you need to specify the item type.

This command creates a new folder `C:\temp\New Folder`:

```
PowerShell  
  
New-Item -Path 'C:\temp\New Folder' -ItemType Directory
```

This command creates a new empty file `C:\temp\New Folder\file.txt`

```
PowerShell  
  
New-Item -Path 'C:\temp\New Folder\file.txt' -ItemType File
```

Important

When using the `Force` switch with the `New-Item` command to create a folder, and the folder already exists, it *won't* overwrite or replace the folder. It will simply return the existing folder object. However, if you use `New-Item -Force` on a file that already exists, the file is overwritten.

Removing all files and folders within a folder

You can remove contained items using `Remove-Item`, but you will be prompted to confirm the removal if the item contains anything else. For example, if you attempt to delete the folder `C:\temp\DeleteMe` that contains other items, PowerShell prompts you for confirmation before deleting the folder:

```
PowerShell  
  
Remove-Item -Path C:\temp\DeleteMe
```

Output

Confirm
The item at `C:\temp\DeleteMe` has children and the `Recurse` parameter wasn't specified. If you continue, all children will be removed with the item. Are you sure you want to continue?

```
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"):
```

If you don't want to be prompted for each contained item, specify the **Recurse** parameter:

```
PowerShell
```

```
Remove-Item -Path C:\temp\DeleteMe -Recurse
```

Mapping a local folder as a drive

You can also map a local folder, using the `New-PSDrive` command. The following command creates a local drive `P:` rooted in the local Program Files directory, visible only from the PowerShell session:

```
PowerShell
```

```
New-PSDrive -Name P -Root $Env:ProgramFiles -PSProvider FileSystem
```

Just as with network drives, drives mapped within PowerShell are immediately visible to the PowerShell shell. To create a mapped drive visible from File Explorer, use the **Persist** parameter. However, only remote paths can be used with **Persist**.

Reading a text file into an array

One of the more common storage formats for text data is in a file with separate lines treated as distinct data elements. The `Get-Content` cmdlet can be used to read an entire file in one step, as shown here:

```
PowerShell
```

```
Get-Content -Path $PROFILE  
# Load modules and change to the PowerShell-Docs repository folder  
Import-Module posh-git  
Set-Location C:\Git\PowerShell-Docs
```

`Get-Content` treats the data read from the file as an array, with one element per line of file content. You can confirm this by checking the **Length** of the returned content:

```
PowerShell
```

```
PS> (Get-Content -Path $PROFILE).Length
```

```
3
```

This command is most useful for getting lists of information into PowerShell. For example, you might store a list of computer names or IP addresses in the file `C:\temp\domainMembers.txt`, with one name on each line of the file. You can use `Get-Content` to retrieve the file contents and put them in the variable `$Computers`:

```
PowerShell
```

```
$Computers = Get-Content -Path C:\temp\DomainMembers.txt
```

`$Computers` is now an array containing a computer name in each element.

Working with files, folders and registry keys

Article • 12/09/2022

This sample only applies to Windows platforms.

PowerShell uses the noun **Item** to refer to items found on a PowerShell drive. When dealing with the PowerShell FileSystem provider, an **Item** might be a file, a folder, or the PowerShell drive. Listing and working with these items is a critical basic task in most administrative settings, so we want to discuss these tasks in detail.

Enumerating files, folders, and registry keys

Since getting a collection of items from a particular location is such a common task, the `Get-ChildItem` cmdlet is designed specifically to return all items found within a container such as a folder.

If you want to return all files and folders that are contained directly within the folder `C:\Windows`, type:

```
PS> Get-ChildItem -Path C:\Windows
Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows

Mode          LastWriteTime    Length Name
----          -----        ---- 
-a---  2006-05-16 8:10 AM      0 0.log
-a---  2005-11-29 3:16 PM     97 acc1.txt
-a---  2005-10-23 11:21 PM   3848 actsetup.log
...
```

The listing looks similar to what you would see when you enter the `dir` command in `cmd.exe`, or the `ls` command in a Unix command shell.

You can perform complex listings using parameters of the `Get-ChildItem` cmdlet. You can see the syntax the `Get-ChildItem` cmdlet by typing:

```
PowerShell
Get-Command -Name Get-ChildItem -Syntax
```

These parameters can be mixed and matched to get highly customized output.

Listing all contained items

To see both the items inside a Windows folder and any items that are contained within the subfolders, use the **Recurse** parameter of `Get-ChildItem`. The listing displays everything within the Windows folder and the items in its subfolders. For example:

```
PS> Get-ChildItem -Path C:\WINDOWS -Recurse

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS\AppPatch
Mode           LastWriteTime      Length Name
----           -----          -----
-a---       2004-08-04   8:00 AM     1852416 AcGenral.dll
...
...
```

Filtering items by name

To display only the names of items, use the **Name** parameter of `Get-ChildItem`:

```
PS> Get-ChildItem -Path C:\WINDOWS -Name
addons
AppPatch
assembly
...
...
```

Forcibly listing hidden items

Items that are hidden in File Explorer or `cmd.exe` aren't displayed in the output of a `Get-ChildItem` command. To display hidden items, use the **Force** parameter of `Get-ChildItem`. For example:

```
PowerShell
Get-ChildItem -Path C:\Windows -Force
```

This parameter is named **Force** because you can forcibly override the normal behavior of the `Get-ChildItem` command. **Force** is a widely used parameter that forces an action

that a cmdlet wouldn't normally perform, although it can't perform any action that compromises the security of the system.

Matching item names with wildcards

The `Get-ChildItem` command accepts wildcards in the path of the items to list.

Because wildcard matching is handled by the PowerShell engine, all cmdlets that accept wildcards use the same notation and have the same matching behavior. The PowerShell wildcard notation includes:

- Asterisk (*) matches zero or more occurrences of any character.
- Question mark (?) matches exactly one character.
- Left bracket ([]) character and right bracket (]) character surround a set of characters to be matched.

Here are some examples of how wildcard specification works.

To find all files in the Windows directory with the suffix `.log` and exactly five characters in the base name, enter the following command:

```
PS> Get-ChildItem -Path C:\Windows\?????.log

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows
Mode           LastWriteTime      Length Name
----           -----          ---- 
...
-a---   2006-05-11  6:31 PM     204276 ocgen.log
-a---   2006-05-11  6:31 PM     22365 ocmsn.log
...
-a---   2005-11-11  4:55 AM       64 setup.log
-a---   2005-12-15  2:24 PM    17719 VxSDM.log
...
```

To find all files that begin with the letter `x` in the Windows directory, type:

```
PowerShell
Get-ChildItem -Path C:\Windows\x*
```

To find all files whose names begin with "x" or "z", type:

```
PowerShell
```

```
Get-ChildItem -Path C:\Windows\[xz]*
```

For more information about wildcards, see [about_Wildcards](#).

Excluding items

You can exclude specific items using the **Exclude** parameter of `Get-ChildItem`. This lets you perform complex filtering in a single statement.

For example, suppose you are trying to find the Windows Time Service DLL in the `System32` folder, and all you can remember about the DLL name is that it begins with "W" and has "32" in it.

An expression like `w*32*.dll` will find all DLLs that satisfy the conditions, but you may want to further filter out the files and omit any `win32` files. You can omit these files using the **Exclude** parameter with the pattern `win*`:

```
PS> Get-ChildItem -Path C:\WINDOWS\System32\w*32*.dll -Exclude win*
```

```
Directory: C:\WINDOWS\System32
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a--	3/18/2019 9:43 PM	495616	w32time.dll
-a--	3/18/2019 9:44 PM	35328	w32topl.dll
-a--	1/24/2020 5:44 PM	401920	Wldap32.dll
-a--	10/10/2019 5:40 PM	442704	ws2_32.dll
-a--	3/18/2019 9:44 PM	66048	wsnmp32.dll
-a--	3/18/2019 9:44 PM	18944	wsock32.dll
-a--	3/18/2019 9:44 PM	64792	wtsapi32.dll

Mixing `Get-ChildItem` parameters

You can use several of the parameters of the `Get-ChildItem` cmdlet in the same command. Before you mix parameters, be sure that you understand wildcard matching. For example, the following command returns no results:

```
PowerShell
```

```
Get-ChildItem -Path C:\Windows\*.dll -Recurse -Exclude [a-y]*.dll
```

There are no results, even though there are two DLLs that begin with the letter "z" in the Windows folder.

No results were returned because we specified the wildcard as part of the path. Even though the command was recursive, the `Get-ChildItem` cmdlet restricted the items to those that are in the Windows folder with names ending with `.dll`.

To specify a recursive search for files whose names match a special pattern, use the **Include** parameter.

```
PS> Get-ChildItem -Path C:\Windows -Include *.dll -Recurse -Exclude [a-y]*.dll

    Directory:
Microsoft.PowerShell.Core\FileSystem::C:\Windows\System32\Setup

Mode           LastWriteTime      Length Name
----           -----          ----- 
-a-- 2004-08-04 8:00 AM        8261 zoneoc.dll

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Windows\System32

Mode           LastWriteTime      Length Name
----           -----          ----- 
-a-- 2004-08-04 8:00 AM       337920 zipfldr.dll
```

Working with registry entries

Article • 07/31/2024

This sample only applies to Windows platforms.

Because registry entries are properties of keys and, as such, can't be directly browsed, we need to take a slightly different approach when working with them.

Listing registry entries

There are many different ways to examine registry entries. The simplest way is to get the property names associated with a key. For example, to see the names of the entries in the registry key `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion`, use `Get-Item`. Registry keys have a property with the generic name of "Property" that's a list of registry entries in the key. The following command selects the `Property` property and expands the items so that they're displayed in a list:

PowerShell

```
Get-Item -Path  
Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion |  
    Select-Object -ExpandProperty Property
```

Output

```
DevicePath  
MediaPathUnexpanded  
ProgramFilesDir  
CommonFilesDir  
ProductId
```

To view the registry entries in a more readable form, use `Get-ItemProperty`:

PowerShell

```
Get-ItemProperty -Path  
Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
```

Output

ProgramFilesDir	: C:\Program Files
CommonFilesDir	: C:\Program Files\Common Files

```
ProgramFilesDir (x86)      : C:\Program Files (x86)
CommonFilesDir (x86)        : C:\Program Files (x86)\Common Files
CommonW6432Dir              : C:\Program Files\Common Files
DevicePath                  : C:\WINDOWS\inf
MediaPathUnexpanded          : C:\WINDOWS\Media
ProgramFilesPath             : C:\Program Files
ProgramW6432Dir              : C:\Program Files
SM_ConfigureProgramsName    : Set Program Access and Defaults
SM_GamesName                 : Games
PSPath                      :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWA
                                RE\Microsoft\Windows\CurrentVersion
PSParentPath                 :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWA
                                RE\Microsoft\Windows
PSChildName                  : CurrentVersion
PSDrive                      : HKLM
PSProvider                   : Microsoft.PowerShell.Core\Registry
```

The Windows PowerShell-related properties for the key are all prefixed with "PS", such as **PSPath**, **PSParentPath**, **PSChildName**, and **PSProvider**.

You can use the `*.*` notation for referring to the current location. You can use `Set-Location` to change to the **CurrentVersion** registry container first:

```
PowerShell

Set-Location -Path
Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
```

Alternatively, you can use the built-in `HKLM:` PSDrive with `Set-Location`:

```
PowerShell

Set-Location -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion
```

You can then use the `.` notation for the current location to list the properties without specifying a full path:

```
PowerShell

Get-ItemProperty -Path .
```

```
Output

...
DevicePath      : C:\WINDOWS\inf
```

```
MediaPathUnexpanded : C:\WINDOWS\Media
ProgramFilesDir      : C:\Program Files
...
```

Path expansion works the same as it does within the filesystem, so from this location you can get the **ItemProperty** listing for `HKLM:\SOFTWARE\Microsoft\Windows\Help` using `Get-ItemProperty -Path ..\Help`.

Getting a single registry entry

If you want to retrieve a specific entry in a registry key, you can use one of several possible approaches. This example finds the value of **DevicePath** in `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion`.

Using `Get-ItemProperty`, use the **Path** parameter to specify the name of the key, and the **Name** parameter to specify the name of the **DevicePath** entry.

PowerShell

```
Get-ItemProperty -Path HKLM:\Software\Microsoft\Windows\CurrentVersion -Name
DevicePath
```

Output

```
DevicePath    : C:\WINDOWS\inf
PSPath        :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\Microsoft\Wi
ndows\CurrentVersion
PSParentPath  :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\Microsoft\Wi
ndows
PSChildName   : CurrentVersion
PSDrive       : HKLM
PSProvider    : Microsoft.PowerShell.Core\Registry
```

This command returns the standard Windows PowerShell properties as well as the **DevicePath** property.

ⓘ Note

Although `Get-ItemProperty` has **Filter**, **Include**, and **Exclude** parameters, they can't be used to filter by property name. These parameters refer to registry keys, which are item paths and not registry entries, which are item properties.

Another option is to use the `reg.exe` command line tool. For help with `reg.exe`, type `reg.exe /?` at a command prompt. To find the **DevicePath** entry, use `reg.exe` as shown in the following command:

PowerShell

```
reg query HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion /v DevicePath
```

Output

```
! REG.EXE VERSION 3.0
```

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion  
    DevicePath  REG_EXPAND_SZ  %SystemRoot%\inf
```

You can also use the **WshShell** COM object to find some registry entries, although this method doesn't work with large binary data or with registry entry names that include characters such as backslash (\). Append the property name to the item path with a \ separator:

PowerShell

```
(New-Object -ComObject  
WScript.Shell).RegRead("HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\DevicePath")
```

Output

```
%SystemRoot%\inf
```

Setting a single registry entry

If you want to change a specific entry in a registry key, you can use one of several possible approaches. This example modifies the **Path** entry under `HKEY_CURRENT_USER\Environment`. The **Path** entry specifies where to find executable files.

1. Retrieve the current value of the **Path** entry using `Get-ItemProperty`.
2. Add the new value, separating it with a ;.
3. Use `Set-ItemProperty` with the specified key, entry name, and value to modify the registry entry.

PowerShell

```
$value = Get-ItemProperty -Path HKCU:\Environment -Name Path  
$newpath = $value.Path += ";C:\src\bin\"  
Set-ItemProperty -Path HKCU:\Environment -Name Path -Value $newpath
```

ⓘ Note

Although `Set-ItemProperty` has `Filter`, `Include`, and `Exclude` parameters, they can't be used to filter by property name. These parameters refer to registry keys—which are item paths—and not registry entries—which are item properties.

Another option is to use the `Reg.exe` command line tool. For help with `reg.exe`, type `reg.exe /?` at a command prompt.

The following example changes the `Path` entry by removing the path added in the example above. `Get-ItemProperty` is still used to retrieve the current value to avoid having to parse the string returned from `reg query`. The `SubString` and `LastIndexOf` methods are used to retrieve the last path added to the `Path` entry.

PowerShell

```
$value = Get-ItemProperty -Path HKCU:\Environment -Name Path  
$newpath = $value.Path.SubString(0, $value.Path.LastIndexOf(';'))  
reg add HKCU\Environment /v Path /d $newpath /f
```

Output

The operation completed successfully.

Creating new registry entries

To add a new entry named "PowerShellPath" to the `CurrentVersion` key, use `New-ItemProperty` with the path to the key, the entry name, and the value of the entry. For this example, we will take the value of the Windows PowerShell variable `$PSHOME`, which stores the path to the installation directory for Windows PowerShell.

You can add the new entry to the key using the following command, and the command also returns information about the new entry:

PowerShell

```
$newItemPropertySplat = @{
    Path = 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion'
    Name = 'PowerShellPath'
    PropertyType = 'String'
    Value = $PSHOME
}
New-ItemProperty @newItemPropertySplat
```

Output

```
PSPath      :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Wi
ndows\CurrentVersion
PSParentPath  :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Wi
ndows
PSChildName   : CurrentVersion
PSDrive       : HKLM
PSProvider     : Microsoft.PowerShell.Core\Registry
PowerShellPath : C:\Program Files\Windows PowerShell\v1.0
```

The **.PropertyType** must be the name of a **Microsoft.Win32.RegistryValueKind** enumeration member from the following table:

- **String** - Used for REG_SZ values. Pass a **[System.String]** object to the **Value** parameter.
- **ExpandString** - Used for REG_EXPAND_SZ values. Pass a **[System.String]** object to the **Value** parameter. The string should contain unexpanded references to environment variables that are expanded when the value is retrieved.
- **Binary** - Used for REG_BINARY values. Pass a **[System.Byte[]]** object to the **Value** parameter.
- **DWord** - Used for REG_DWORD values. Pass a **[System.Int32]** object to the **Value** parameter.
- **MultiString** - Used for REG_MULTI_SZ values. Pass a **[System.String[]]** object to the **Value** parameter.
- **QWord** - Used for REG_QWORD values. Pass a **[System.Int64]** object to the **Value** parameter.

You can add a registry entry to multiple locations by specifying an array of values for the **Path** parameter:

PowerShell

```
$newItemPropertySplat = @{
    Name = 'PowerShellPath'
```

```
.PropertyType = 'String'
Value = $PSHOME
Path = 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion',
      'HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion'
}
New-ItemProperty @newItemPropertySplat
```

You can also overwrite a pre-existing registry entry value by adding the **Force** parameter to any `New-ItemProperty` command.

The following examples show how to create new registry entries of various types. The registry values are created in a new key named **MySoftwareKey** under `HKEY_CURRENT_USER\Software`. The `$key` variable is used to store the new key object.

PowerShell

```
$key = New-Item -Path HKCU:\Software -Name MySoftwareKey
$newItemPropertySplat = @{
    Path = $key.PSPath
    Name = 'DefaultFolders'
    PropertyType = 'MultiString'
    Value = 'Home', 'Temp', 'Publish'
}
New-ItemProperty @newItemPropertySplat
```

Output

```
DefaultFolders : {Home, Temp, Publish}
PSPath         :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\MySoftwareKey
PSParentPath   :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software
PSChildName   : MySoftwareKey
PSProvider     : Microsoft.PowerShell.Core\Registry
```

You can use the **PSPath** property of the key object in subsequent commands.

PowerShell

```
New-ItemProperty -Path $key.PSPath -Name MaxAllowed -PropertyType QWord -
Value 1024
```

Output

```
MaxAllowed   : 1024
PSPath       :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\MySoftwareKey
```

```
PSParentPath :  
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software  
PSChildName  : MySoftwareKey  
PSPrinter     : Microsoft.PowerShell.Core\Registry
```

You can also pipe `$key` to `New-ItemProperty` to add a value to the key.

PowerShell

```
$date = Get-Date -Format 'dd-MMM-yyyy'  
$newItemPropertySplat = @{  
    Name = 'BinaryDate'  
    PropertyType = 'Binary'  
    Value = ([System.Text.Encoding]::UTF8.GetBytes($date))  
}  
$key | New-ItemProperty @newItemPropertySplat
```

Output

```
BinaryDate    : {51, 49, 45, 74...}  
PSPath        :  
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\MySoftwareKey  
PSParentPath :  
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software  
PSChildName  : MySoftwareKey  
PSPrinter     : Microsoft.PowerShell.Core\Registry
```

Displaying the content of `$key` shows the new entries.

PowerShell

```
$key
```

Output

```
Hive: HKEY_CURRENT_USER\Software
```

Name	Property
---	-----
MySoftwareKey	DefaultFolders : {Home, Temp, Publish} MaxAllowed : 1024 BinaryDate : {51, 49, 45, 74...}

The following example shows the value type for each kind of registry entry:

PowerShell

```
$key.GetValueNames() | Select-Object @{n='ValueName';e={$_.}},  
@{n='ValueKind';e={$key.GetValueKind($_)}},  
@{n='Type';e={$key.GetValue($_).GetType()}},  
@{n='Value';e={$key.GetValue($_)}}
```

Output

ValueName	ValueKind	Type	Value
DefaultFolders	MultiString	System.String[]	{Home, Temp, Publish}
MaxAllowed	QWord	System.Int64	1024
BinaryDate	Binary	System.Byte[]	{51, 49, 45, 74...}

Renaming registry entries

To rename the **PowerShellPath** entry to "PSHome," use `Rename-ItemProperty`:

PowerShell

```
$renameItemPropertySplat = @{  
    Path = 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion'  
    Name = 'PowerShellPath'  
    NewName = 'PSHome'  
}  
Rename-ItemProperty @renameItemPropertySplat
```

To display the renamed value, add the **PassThru** parameter to the command.

PowerShell

```
$renameItemPropertySplat = @{  
    Path = 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion'  
    Name = 'PowerShellPath'  
    NewName = 'PSHome'  
    PassThru = $true  
}  
Rename-ItemProperty @renameItemPropertySplat
```

Deleting registry entries

To delete both the PSHome and PowerShellPath registry entries, use `Remove-ItemProperty`:

PowerShell

```
Remove-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion -  
Name PSHome  
Remove-ItemProperty -Path HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion -  
Name PowerShellPath
```

Working with registry keys

Article • 12/09/2022

This sample only applies to Windows platforms.

Because registry keys are items on PowerShell drives, working with them is very similar to working with files and folders. One critical difference is that every item on a registry-based PowerShell drive is a container, just like a folder on a file system drive. However, registry entries and their associated values are properties of the items, not distinct items.

Listing all subkeys of a registry key

You can show all items directly within a registry key using `Get-ChildItem`. Add the optional **Force** parameter to display hidden or system items. For example, this command displays the items directly within PowerShell drive `HKCU:`, which corresponds to the `HKEY_CURRENT_USER` registry hive:

PowerShell

```
Get-ChildItem -Path HKCU:\ | Select-Object Name
```

Output

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

Name

HKEY_CURRENT_USER\AppEvents
HKEY_CURRENT_USER\Console
HKEY_CURRENT_USER\Control Panel
HKEY_CURRENT_USER\DirectShow
HKEY_CURRENT_USER\dummy
HKEY_CURRENT_USER\Environment
HKEY_CURRENT_USER\EUDC
HKEY_CURRENT_USER\Keyboard Layout
HKEY_CURRENT_USER\MediaFoundation
HKEY_CURRENT_USER\Microsoft
HKEY_CURRENT_USER\Network
HKEY_CURRENT_USER\Printers
HKEY_CURRENT_USER\Software
HKEY_CURRENT_USER\System
HKEY_CURRENT_USER\Uninstall
HKEY_CURRENT_USER\WXP
HKEY_CURRENT_USER\Volatile Environment

These are the top-level keys visible under `HKEY_CURRENT_USER` in the Registry Editor (`regedit.exe`).

You can also specify this registry path by specifying the Registry provider's name, followed by `:::`. The Registry provider's full name is `Microsoft.PowerShell.Core\Registry`, but this can be shortened to just `Registry`. Any of the following commands will list the contents directly under `HKCU::`.

PowerShell

```
Get-ChildItem -Path Registry::HKEY_CURRENT_USER
Get-ChildItem -Path Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
Get-ChildItem -Path Registry::HKCU
Get-ChildItem -Path Microsoft.PowerShell.Core\Registry::HKCU
Get-ChildItem HKCU:
```

These commands list only the directly contained items, much like using `DIR` in `cmd.exe` or `ls` in a Unix shell. To show contained items, you need to specify the `Recurse` parameter. To list all registry keys in `HKCU::`, use the following command.

PowerShell

```
Get-ChildItem -Path HKCU:\ -Recurse
```

`Get-ChildItem` can perform complex filtering capabilities through its `Path`, `Filter`, `Include`, and `Exclude` parameters, but those parameters are typically based only on name. You can perform complex filtering based on other properties of items using the `Where-Object` cmdlet. The following command finds all keys within `HKCU:\Software` that have no more than one subkey and also have exactly four values:

PowerShell

```
Get-ChildItem -Path HKCU:\Software -Recurse |
  Where-Object {($_.SubKeyCount -le 1) -and ($_.ValueCount -eq 4) }
```

Copying keys

Copying is done with `Copy-Item`. The following example copies the `CurrentVersion` subkey of `HKLM:\SOFTWARE\Microsoft\Windows\` and all of its properties to `HKCU:\`.

PowerShell

```
Copy-Item -Path 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion' -  
Destination HKCU:
```

If you examine this new key in the registry editor or using `Get-ChildItem`, you notice that you don't have copies of the contained subkeys in the new location. In order to copy all of the contents of a container, you need to specify the `Recurse` parameter. To make the preceding copy command recursive, you would use this command:

PowerShell

```
Copy-Item -Path 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion' -  
Destination HKCU: -Recurse
```

You can still use other tools you already have available to perform filesystem copies. Any registry editing tools—including `reg.exe`, `regini.exe`, `regedit.exe`, and COM objects that support registry editing, such as `WScript.Shell` and WMI's `StdRegProv` class can be used from within PowerShell.

Creating keys

Creating new keys in the registry is simpler than creating a new item in a file system. Because all registry keys are containers, you don't need to specify the item type. Just provide an explicit path, such as:

PowerShell

```
New-Item -Path HKCU:\Software_DeleteMe
```

You can also use a provider-based path to specify a key:

PowerShell

```
New-Item -Path Registry::HKCU\Software_DeleteMe
```

Deleting keys

Deleting items is essentially the same for all providers. The following commands silently remove items:

PowerShell

```
Remove-Item -Path HKCU:\Software_DeleteMe
Remove-Item -Path 'HKCU:\key with spaces in the name'
```

Removing all keys under a specific key

You can remove contained items using `Remove-Item`, but you will be prompted to confirm the removal if the item contains anything else. For example, if we attempt to delete the `HKCU:\CurrentVersion` subkey we created, we see this:

PowerShell

```
Remove-Item -Path HKCU:\CurrentVersion
```

Output

Confirm

The item at `HKCU:\CurrentVersion\AdminDebug` has children and the `Recurse` parameter was not specified. If you continue, all children will be removed with

the item. Are you sure you want to continue?

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):

To delete contained items without prompting, specify the `Recurse` parameter:

PowerShell

```
Remove-Item -Path HKCU:\CurrentVersion -Recurse
```

If you wanted to remove all items within `HKCU:\CurrentVersion` but not `HKCU:\CurrentVersion` itself, you could instead use:

PowerShell

```
Remove-Item -Path HKCU:\CurrentVersion\* -Recurse
```

Creating a custom input box

Article • 12/09/2022

This sample only applies to Windows platforms.

Script a graphical custom input box using Microsoft .NET Framework form-building features in Windows PowerShell 3.0 and later releases.

Create a custom, graphical input box

Copy and then paste the following into Windows PowerShell ISE, and then save it as a PowerShell script (.ps1) file.

PowerShell

```
Add-Type -AssemblyName System.Windows.Forms
Add-Type -AssemblyName System.Drawing

$form = New-Object System.Windows.Forms.Form
$form.Text = 'Data Entry Form'
$form.Size = New-Object System.Drawing.Size(300,200)
$form.StartPosition = 'CenterScreen'

$okButton = New-Object System.Windows.Forms.Button
$okButton.Location = New-Object System.Drawing.Point(75,120)
$okButton.Size = New-Object System.Drawing.Size(75,23)
$okButton.Text = 'OK'
$okButton.DialogResult = [System.Windows.Forms.DialogResult]::OK
$form.AcceptButton = $okButton
$form.Controls.Add($okButton)

$cancelButton = New-Object System.Windows.Forms.Button
$cancelButton.Location = New-Object System.Drawing.Point(150,120)
$cancelButton.Size = New-Object System.Drawing.Size(75,23)
$cancelButton.Text = 'Cancel'
$cancelButton.DialogResult = [System.Windows.Forms.DialogResult]::Cancel
$form.CancelButton = $cancelButton
$form.Controls.Add($cancelButton)

$label = New-Object System.Windows.Forms.Label
$label.Location = New-Object System.Drawing.Point(10,20)
$label.Size = New-Object System.Drawing.Size(280,20)
$label.Text = 'Please enter the information in the space below:'
$form.Controls.Add($label)

$textBox = New-Object System.Windows.Forms.TextBox
$textBox.Location = New-Object System.Drawing.Point(10,40)
$textBox.Size = New-Object System.Drawing.Size(260,20)
```

```
$form.Controls.Add($textBox)

$form.Topmost = $true

$form.Add_Shown({$textBox.Select()})
$result = $form.ShowDialog()

if ($result -eq [System.Windows.Forms.DialogResult]::OK)
{
    $x = $textBox.Text
    $x
}
```

The script begins by loading two .NET Framework classes: **System.Drawing** and **System.Windows.Forms**. You then start a new instance of the .NET Framework class **System.Windows.Forms.Form**. That provides a blank form or window to which you can start adding controls.

PowerShell

```
$form = New-Object System.Windows.Forms.Form
```

After you create an instance of the Form class, assign values to three properties of this class.

- **Text**. This becomes the title of the window.
- **Size**. This is the size of the form, in pixels. The preceding script creates a form that's 300 pixels wide by 200 pixels tall.
- **StartPosition**. This optional property is set to **CenterScreen** in the preceding script. If you don't add this property, Windows selects a location when the form is opened. By setting the **StartPosition** to **CenterScreen**, you're automatically displaying the form in the middle of the screen each time it loads.

PowerShell

```
$form.Text = 'Data Entry Form'
$form.Size = New-Object System.Drawing.Size(300,200)
$form.StartPosition = 'CenterScreen'
```

Next, create an **OK** button for your form. Specify the size and behavior of the **OK** button. In this example, the button position is 120 pixels from the form's top edge, and 75 pixels from the left edge. The button height is 23 pixels, while the button length is 75 pixels. The script uses predefined Windows Forms types to determine the button behaviors.

PowerShell

```
$okButton = New-Object System.Windows.Forms.Button
$okButton.Location = New-Object System.Drawing.Point(75,120)
$okButton.Size = New-Object System.Drawing.Size(75,23)
$okButton.Text = 'OK'
$okButton.DialogResult = [System.Windows.Forms.DialogResult]::OK
$form.AcceptButton = $okButton
$form.Controls.Add($okButton)
```

Similarly, you create a **Cancel** button. The **Cancel** button is 120 pixels from the top, but 150 pixels from the left edge of the window.

PowerShell

```
$cancelButton = New-Object System.Windows.Forms.Button
$cancellationToken.Location = New-Object System.Drawing.Point(150,120)
$cancellationToken.Size = New-Object System.Drawing.Size(75,23)
$cancellationToken.Text = 'Cancel'
$cancellationToken.DialogResult = [System.Windows.Forms.DialogResult]::Cancel
$form.CancelButton = $cancelButton
$form.Controls.Add($cancelButton)
```

Next, provide label text on your window that describes the information you want users to provide.

PowerShell

```
$label = New-Object System.Windows.Forms.Label
$label.Location = New-Object System.Drawing.Point(10,20)
$label.Size = New-Object System.Drawing.Size(280,20)
$label.Text = 'Please enter the information in the space below:'
$form.Controls.Add($label)
```

Add the control (in this case, a text box) that lets users provide the information you've described in your label text. There are many other controls you can apply besides text boxes. For more controls, see [System.Windows.Forms Namespace](#).

PowerShell

```
$textBox = New-Object System.Windows.Forms.TextBox
$textBox.Location = New-Object System.Drawing.Point(10,40)
$textBox.Size = New-Object System.Drawing.Size(260,20)
$form.Controls.Add($textBox)
```

Set the **Topmost** property to **\$true** to force the window to open atop other open windows and dialog boxes.

```
PowerShell
```

```
$form.Topmost = $true
```

Next, add this line of code to activate the form, and set the focus to the text box that you created.

```
PowerShell
```

```
$form.Add_Shown({$textBox.Select()})
```

Add the following line of code to display the form in Windows.

```
PowerShell
```

```
$result = $form.ShowDialog()
```

Finally, the code inside the `if` block instructs Windows what to do with the form after users provide text in the text box, and then click the **OK** button or press the **Enter** key.

```
PowerShell
```

```
if ($result -eq [System.Windows.Forms.DialogResult]::OK) {  
    $x = $textBox.Text  
    $x  
}
```

See also

- [GitHub: Dave Wyatt's WinFormsExampleUpdates ↗](#)
- [Windows PowerShell Tip of the Week: Creating a Custom Input Box](#)

Creating a graphical date picker

Article • 12/09/2022

This sample only applies to Windows platforms.

Use Windows PowerShell 3.0 and later releases to create a form with a graphical, calendar-style control that lets users select a day of the month.

Create a graphical date-picker control

Copy and then paste the following into Windows PowerShell ISE, and then save it as a PowerShell script (.ps1) file.

PowerShell

```
Add-Type -AssemblyName System.Windows.Forms
Add-Type -AssemblyName System.Drawing

$form = New-Object Windows.Forms.Form -Property @{
    StartPosition = [Windows.Forms.FormStartPosition]::CenterScreen
    Size          = New-Object Drawing.Size 243, 230
    Text          = 'Select a Date'
    Topmost       = $true
}

$calendar = New-Object Windows.Forms.MonthCalendar -Property @{
    ShowTodayCircle = $false
    MaxSelectionCount = 1
}
$form.Controls.Add($calendar)

$okButton = New-Object Windows.Forms.Button -Property @{
    Location      = New-Object Drawing.Point 38, 165
    Size          = New-Object Drawing.Size 75, 23
    Text          = 'OK'
    DialogResult = [Windows.Forms.DialogResult]::OK
}
$form.AcceptButton = $okButton
$form.Controls.Add($okButton)

$cancelButton = New-Object Windows.Forms.Button -Property @{
    Location      = New-Object Drawing.Point 113, 165
    Size          = New-Object Drawing.Size 75, 23
    Text          = 'Cancel'
    DialogResult = [Windows.Forms.DialogResult]::Cancel
}
$form.CancelButton = $cancelButton
$form.Controls.Add($cancelButton)
```

```
$result = $form.ShowDialog()

if ($result -eq [Windows.Forms.DialogResult]::OK) {
    $date = $calendar.SelectionStart
    Write-Host "Date selected: $($date.ToString())"
}
```

The script begins by loading two .NET Framework classes: `System.Drawing` and `System.Windows.Forms`. You then start a new instance of the .NET Framework class `Windows.Forms.Form`. That provides a blank form or window to which you can start adding controls.

PowerShell

```
$form = New-Object Windows.Forms.Form -Property @{
    StartPosition = [Windows.Forms.FormStartPosition]::CenterScreen
    Size          = New-Object Drawing.Size 243, 230
    Text          = 'Select a Date'
    Topmost       = $true
}
```

This example assigns values to four properties of this class by using the `Property` property and hashtable.

1. **StartPosition:** If you don't add this property, Windows selects a location when the form is opened. By setting this property to `CenterScreen`, you're automatically displaying the form in the middle of the screen each time it loads.
2. **Size:** This is the size of the form, in pixels. The preceding script creates a form that's 243 pixels wide by 230 pixels tall.
3. **Text:** This becomes the title of the window.
4. **Topmost:** By setting this property to `$true`, you can force the window to open atop other open windows and dialog boxes.

Next, create and then add a calendar control in your form. In this example, the current day isn't highlighted or circled. Users can select only one day on the calendar at one time.

PowerShell

```
$calendar = New-Object Windows.Forms.MonthCalendar -Property @{
    ShowTodayCircle = $false
    MaxSelectionCount = 1
```

```
}
```

```
$form.Controls.Add($calendar)
```

Next, create an **OK** button for your form. Specify the size and behavior of the **OK** button. In this example, the button position is 165 pixels from the form's top edge, and 38 pixels from the left edge. The button height is 23 pixels, while the button length is 75 pixels. The script uses predefined Windows Forms types to determine the button behaviors.

PowerShell

```
$okButton = New-Object Windows.Forms.Button -Property @{
    Location      = New-Object Drawing.Point 38, 165
    Size          = New-Object Drawing.Size 75, 23
    Text          = 'OK'
    DialogResult = [Windows.Forms.DialogResult]::OK
}
$form.AcceptButton = $okButton
$form.Controls.Add($okButton)
```

Similarly, you create a **Cancel** button. The **Cancel** button is 165 pixels from the top, but 113 pixels from the left edge of the window.

PowerShell

```
$cancelButton = New-Object Windows.Forms.Button -Property @{
    Location      = New-Object Drawing.Point 113, 165
    Size          = New-Object Drawing.Size 75, 23
    Text          = 'Cancel'
    DialogResult = [Windows.Forms.DialogResult]::Cancel
}
$form.CancelButton = $cancelButton
$form.Controls.Add($cancelButton)
```

Add the following line of code to display the form in Windows.

PowerShell

```
$result = $form.ShowDialog()
```

Finally, the code inside the `if` block instructs Windows what to do with the form after users select a day on the calendar, and then click the **OK** button or press the **Enter** key. Windows PowerShell displays the selected date to users.

PowerShell

```
if ($result -eq [Windows.Forms.DialogResult]::OK) {  
    $date = $calendar.SelectionStart  
    Write-Host "Date selected: $($date.ToString())"  
}
```

See also

- GitHub: Dave Wyatt's WinFormsExampleUpdates ↗
- Windows PowerShell Tip of the Week: Creating a Graphical Date Picker)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Multiple-selection list boxes

Article • 12/09/2022

This sample only applies to Windows platforms.

Use Windows PowerShell 3.0 and later releases to create a multiple-selection list box control in a custom Windows Form.

Create list box controls that allow multiple selections

Copy and then paste the following into Windows PowerShell ISE, and then save it as a PowerShell script (.ps1) file.

PowerShell

```
Add-Type -AssemblyName System.Windows.Forms
Add-Type -AssemblyName System.Drawing

$form = New-Object System.Windows.Forms.Form
$form.Text = 'Data Entry Form'
$form.Size = New-Object System.Drawing.Size(300,200)
$form.StartPosition = 'CenterScreen'

$OKButton = New-Object System.Windows.Forms.Button
$OKButton.Location = New-Object System.Drawing.Point(75,120)
$OKButton.Size = New-Object System.Drawing.Size(75,23)
$OKButton.Text = 'OK'
$OKButton.DialogResult = [System.Windows.Forms.DialogResult]::OK
$form.AcceptButton = $OKButton
$form.Controls.Add($OKButton)

$CancelButton = New-Object System.Windows.Forms.Button
$CancelButton.Location = New-Object System.Drawing.Point(150,120)
$CancelButton.Size = New-Object System.Drawing.Size(75,23)
$CancelButton.Text = 'Cancel'
$CancelButton.DialogResult = [System.Windows.Forms.DialogResult]::Cancel
$form.CancelButton = $CancelButton
$form.Controls.Add($CancelButton)

$label = New-Object System.Windows.Forms.Label
$label.Location = New-Object System.Drawing.Point(10,20)
$label.Size = New-Object System.Drawing.Size(280,20)
$label.Text = 'Please make a selection from the list below:'
$form.Controls.Add($label)

$listBox = New-Object System.Windows.Forms.ListBox
$listBox.Location = New-Object System.Drawing.Point(10,50)
$listBox.Size = New-Object System.Drawing.Size(280,150)
$listBox.SelectionMode = 'MultipleExtended'
$form.Controls.Add($listBox)
```

```

$listBox.Location = New-Object System.Drawing.Point(10,40)
$listBox.Size = New-Object System.Drawing.Size(260,20)

$listBox.SelectionMode = 'MultiExtended'

[void] $listBox.Items.Add('Item 1')
[void] $listBox.Items.Add('Item 2')
[void] $listBox.Items.Add('Item 3')
[void] $listBox.Items.Add('Item 4')
[void] $listBox.Items.Add('Item 5')

$listBox.Height = 70
$form.Controls.Add($listBox)
$form.Topmost = $true

$result = $form.ShowDialog()

if ($result -eq [System.Windows.Forms.DialogResult]::OK)
{
    $x = $listBox.SelectedItems
    $x
}

```

The script begins by loading two .NET Framework classes: `System.Drawing` and `System.Windows.Forms`. You then start a new instance of the .NET Framework class `System.Windows.Forms.Form`. That provides a blank form or window to which you can start adding controls.

PowerShell

```
$form = New-Object System.Windows.Forms.Form
```

After you create an instance of the `Form` class, assign values to three properties of this class.

- **Text**. This becomes the title of the window.
- **Size**. This is the size of the form, in pixels. The preceding script creates a form that's 300 pixels wide by 200 pixels tall.
- **StartPosition**. This optional property is set to `CenterScreen` in the preceding script. If you don't add this property, Windows selects a location when the form is opened. By setting the `StartPosition` to `CenterScreen`, you're automatically displaying the form in the middle of the screen each time it loads.

PowerShell

```
$form.Text = 'Data Entry Form'
$form.Size = New-Object System.Drawing.Size(300,200)
```

```
$form.StartPosition = 'CenterScreen'
```

Next, create an **OK** button for your form. Specify the size and behavior of the **OK** button. In this example, the button position is 120 pixels from the form's top edge, and 75 pixels from the left edge. The button height is 23 pixels, while the button length is 75 pixels. The script uses predefined Windows Forms types to determine the button behaviors.

PowerShell

```
$OKButton = New-Object System.Windows.Forms.Button
$OKButton.Location = New-Object System.Drawing.Size(75,120)
$OKButton.Size = New-Object System.Drawing.Size(75,23)
$OKButton.Text = 'OK'
$OKButton.DialogResult = [System.Windows.Forms.DialogResult]::OK
$form.AcceptButton = $OKButton
$form.Controls.Add($OKButton)
```

Similarly, you create a **Cancel** button. The **Cancel** button is 120 pixels from the top, but 150 pixels from the left edge of the window.

PowerShell

```
$CancelButton = New-Object System.Windows.Forms.Button
$CancelButton.Location = New-Object System.Drawing.Point(150,120)
$CancelButton.Size = New-Object System.Drawing.Size(75,23)
$CancelButton.Text = 'Cancel'
$CancelButton.DialogResult = [System.Windows.Forms.DialogResult]::Cancel
$form.CancelButton = $CancelButton
$form.Controls.Add($CancelButton)
```

Next, provide label text on your window that describes the information you want users to provide.

PowerShell

```
$label = New-Object System.Windows.Forms.Label
$label.Location = New-Object System.Drawing.Point(10,20)
$label.Size = New-Object System.Drawing.Size(280,20)
$label.Text = 'Please make a selection from the list below:'
$form.Controls.Add($label)
```

Add the control (in this case, a list box) that lets users provide the information you've described in your label text. There are many other controls you can apply besides text boxes; for more controls, see [System.Windows.Forms Namespace](#).

PowerShell

```
$listBox = New-Object System.Windows.Forms.ListBox  
$listBox.Location = New-Object System.Drawing.Point(10,40)  
$listBox.Size = New-Object System.Drawing.Size(260,20)
```

Here's how you specify that you want to allow users to select multiple values from the list.

PowerShell

```
$listBox.SelectionMode = 'MultiExtended'
```

In the next section, you specify the values you want the list box to display to users.

PowerShell

```
[void] $listBox.Items.Add('Item 1')  
[void] $listBox.Items.Add('Item 2')  
[void] $listBox.Items.Add('Item 3')  
[void] $listBox.Items.Add('Item 4')  
[void] $listBox.Items.Add('Item 5')
```

Specify the maximum height of the list box control.

PowerShell

```
$listBox.Height = 70
```

Add the list box control to your form, and instruct Windows to open the form atop other windows and dialog boxes when it's opened.

PowerShell

```
$form.Controls.Add($listBox)  
$form.Topmost = $true
```

Add the following line of code to display the form in Windows.

PowerShell

```
$result = $form.ShowDialog()
```

Finally, the code inside the `if` block instructs Windows what to do with the form after users select one or more options from the list box, and then click the **OK** button or press

the **Enter** key.

PowerShell

```
if ($result -eq [System.Windows.Forms.DialogResult]::OK)
{
    $x = $listBox.SelectedItems
    $x
}
```

See also

- [Weekend Scripter: Fixing PowerShell GUI Examples ↗](#)
- [GitHub: Dave Wyatt's WinFormsExampleUpdates ↗](#)
- [Windows PowerShell Tip of the Week: Multi-Select List Boxes - And More!\)](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Selecting items from a list box

Article • 12/09/2022

This sample only applies to Windows platforms.

Use Windows PowerShell 3.0 and later releases to create a dialog box that lets users select items from a list box control.

Create a list box control, and select items from it

Copy and then paste the following into Windows PowerShell ISE, and then save it as a PowerShell script (.ps1) file.

PowerShell

```
Add-Type -AssemblyName System.Windows.Forms
Add-Type -AssemblyName System.Drawing

$form = New-Object System.Windows.Forms.Form
$form.Text = 'Select a Computer'
$form.Size = New-Object System.Drawing.Size(300,200)
$form.StartPosition = 'CenterScreen'

$okButton = New-Object System.Windows.Forms.Button
$okButton.Location = New-Object System.Drawing.Point(75,120)
$okButton.Size = New-Object System.Drawing.Size(75,23)
$okButton.Text = 'OK'
$okButton.DialogResult = [System.Windows.Forms.DialogResult]::OK
$form.AcceptButton = $okButton
$form.Controls.Add($okButton)

$cancelButton = New-Object System.Windows.Forms.Button
$cancelButton.Location = New-Object System.Drawing.Point(150,120)
$cancelButton.Size = New-Object System.Drawing.Size(75,23)
$cancelButton.Text = 'Cancel'
$cancelButton.DialogResult = [System.Windows.Forms.DialogResult]::Cancel
$form.CancelButton = $cancelButton
$form.Controls.Add($cancelButton)

$label = New-Object System.Windows.Forms.Label
$label.Location = New-Object System.Drawing.Point(10,20)
$label.Size = New-Object System.Drawing.Size(280,20)
$label.Text = 'Please select a computer:'
$form.Controls.Add($label)

$listBox = New-Object System.Windows.Forms.ListBox
```

```

$listBox.Location = New-Object System.Drawing.Point(10,40)
$listBox.Size = New-Object System.Drawing.Size(260,20)
$listBox.Height = 80

[void] $listBox.Items.Add('atl-dc-001')
[void] $listBox.Items.Add('atl-dc-002')
[void] $listBox.Items.Add('atl-dc-003')
[void] $listBox.Items.Add('atl-dc-004')
[void] $listBox.Items.Add('atl-dc-005')
[void] $listBox.Items.Add('atl-dc-006')
[void] $listBox.Items.Add('atl-dc-007')

$form.Controls.Add($listBox)

$form.Topmost = $true

$result = $form.ShowDialog()

if ($result -eq [System.Windows.Forms.DialogResult]::OK)
{
    $x = $listBox.SelectedItem
    $x
}

```

The script begins by loading two .NET Framework classes: **System.Drawing** and **System.Windows.Forms**. You then start a new instance of the .NET Framework class **System.Windows.Forms.Form**. That provides a blank form or window to which you can start adding controls.

PowerShell

```

Add-Type -AssemblyName System.Windows.Forms
Add-Type -AssemblyName System.Drawing

```

After you create an instance of the **Form** class, assign values to three properties of this class.

- **Text**. This becomes the title of the window.
- **Size**. This is the size of the form, in pixels. The preceding script creates a form that's 300 pixels wide by 200 pixels tall.
- **StartPosition**. This optional property is set to **CenterScreen** in the preceding script. If you don't add this property, Windows selects a location when the form is opened. By setting the **StartPosition** to **CenterScreen**, you're automatically displaying the form in the middle of the screen each time it loads.

PowerShell

```
$form.Text = 'Select a Computer'  
$form.Size = New-Object System.Drawing.Size(300,200)  
$form.StartPosition = 'CenterScreen'
```

Next, create an **OK** button for your form. Specify the size and behavior of the **OK** button. In this example, the button position is 120 pixels from the form's top edge, and 75 pixels from the left edge. The button height is 23 pixels, while the button length is 75 pixels. The script uses predefined Windows Forms types to determine the button behaviors.

PowerShell

```
$okButton = New-Object System.Windows.Forms.Button  
$okButton.Location = New-Object System.Drawing.Point(75,120)  
$okButton.Size = New-Object System.Drawing.Size(75,23)  
$okButton.Text = 'OK'  
$okButton.DialogResult = [System.Windows.Forms.DialogResult]::OK  
$form.AcceptButton = $okButton  
$form.Controls.Add($okButton)
```

Similarly, you create a **Cancel** button. The **Cancel** button is 120 pixels from the top, but 150 pixels from the left edge of the window.

PowerShell

```
$cancelButton = New-Object System.Windows.Forms.Button  
$cancelButton.Location = New-Object System.Drawing.Point(150,120)  
$cancelButton.Size = New-Object System.Drawing.Size(75,23)  
$cancelButton.Text = 'Cancel'  
$cancelButton.DialogResult = [System.Windows.Forms.DialogResult]::Cancel  
$form.CancelButton = $cancelButton  
$form.Controls.Add($cancelButton)
```

Next, provide label text on your window that describes the information you want users to provide. In this case, you want users to select a computer.

PowerShell

```
$label = New-Object System.Windows.Forms.Label  
$label.Location = New-Object System.Drawing.Point(10,20)  
$label.Size = New-Object System.Drawing.Size(280,20)  
$label.Text = 'Please select a computer:'  
$form.Controls.Add($label)
```

Add the control (in this case, a list box) that lets users provide the information you've described in your label text. There are many other controls you can apply besides list boxes; for more controls, see [System.Windows.Forms Namespace](#).

PowerShell

```
$listBox = New-Object System.Windows.Forms.ListBox  
$listBox.Location = New-Object System.Drawing.Point(10,40)  
$listBox.Size = New-Object System.Drawing.Size(260,20)  
$listBox.Height = 80
```

In the next section, you specify the values you want the list box to display to users.

ⓘ Note

The list box created by this script allows only one selection. To create a list box control that allows multiple selections, specify a value for the **SelectionMode** property, similarly to the following: `$listBox.SelectionMode = 'MultiExtended'`. For more information, see [Multiple-selection List Boxes](#).

PowerShell

```
[void] $listBox.Items.Add('atl-dc-001')  
[void] $listBox.Items.Add('atl-dc-002')  
[void] $listBox.Items.Add('atl-dc-003')  
[void] $listBox.Items.Add('atl-dc-004')  
[void] $listBox.Items.Add('atl-dc-005')  
[void] $listBox.Items.Add('atl-dc-006')  
[void] $listBox.Items.Add('atl-dc-007')
```

Add the list box control to your form, and instruct Windows to open the form atop other windows and dialog boxes when it's opened.

PowerShell

```
$form.Controls.Add($listBox)  
$form.Topmost = $true
```

Add the following line of code to display the form in Windows.

PowerShell

```
$result = $form.ShowDialog()
```

Finally, the code inside the `if` block instructs Windows what to do with the form after users select an option from the list box, and then click the **OK** button or press the **Enter** key.

PowerShell

```
if ($result -eq [System.Windows.Forms.DialogResult]::OK) {  
    $x = $listBox.SelectedItem  
    $x  
}
```

See also

- [GitHub: Dave Wyatt's WinFormsExampleUpdates ↗](#)
- [Windows PowerShell Tip of the Week: Selecting Items from a List Box](#)

Using Experimental Features in PowerShell

Article • 01/23/2025

The Experimental Features support in PowerShell provides a mechanism for experimental features to coexist with existing stable features in PowerShell or PowerShell modules.

An experimental feature is one where the design isn't finalized. The feature is available for users to test and provide feedback. Once an experimental feature is finalized, the design changes become breaking changes.

✖ Caution

Experimental features aren't intended to be used in production since the changes are allowed to be breaking. Experimental features aren't officially supported. However, we appreciate any feedback and bug reports. You can file issues in the [GitHub source repository](#).

For more information about enabling or disabling these features, see [about_Experimental_Features](#).

Experimental feature lifecycle

The `Get-ExperimentalFeature` cmdlet returns all experimental features available to PowerShell. Experimental features can come from modules or the PowerShell engine. Module-based experimental features are only available after you import the module. In the following example, the `PSDesiredStateConfiguration` isn't loaded, so the `PSDesiredStateConfiguration.InvokeDscResource` feature isn't available.

PowerShell	Output								
<pre>Get-ExperimentalFeature</pre>									
	<table><thead><tr><th>Name</th><th>Enabled</th><th>Source</th><th>Description</th></tr></thead><tbody><tr><td>PSCommandNotFoundSuggestion</td><td>False</td><td>PSEngine</td><td>Recommend potential commands based on fuzzy searc...</td></tr></tbody></table>	Name	Enabled	Source	Description	PSCommandNotFoundSuggestion	False	PSEngine	Recommend potential commands based on fuzzy searc...
Name	Enabled	Source	Description						
PSCommandNotFoundSuggestion	False	PSEngine	Recommend potential commands based on fuzzy searc...						

PSCommandWithArgs	False	PSEngine Enable ``-CommandWithArgs`` parameter for pwsh
PSFeedbackProvider	True	PSEngine Replace the hard-coded suggestion framework with ...
PSLoadAssemblyFromNativeCode	False	PSEngine Expose an API to allow assembly loading from native...
PSModuleAutoLoadSkipOfflineFiles	True	PSEngine Module discovery will skip over files that are ma...
PSSerializeJSONLongEnumAsNumber	True	PSEngine Serialize enums based on long or ulong as an nume...
PSSubsystemPluginModel	True	PSEngine A plugin model for registering and un-registering...

Use the [Enable-ExperimentalFeature](#) and [Disable-ExperimentalFeature](#) cmdlets to enable or disable a feature. You must start a new PowerShell session for this change to be in effect. Run the following command to enable the `PSCommandNotFoundSuggestion` feature:

PowerShell

```
Enable-ExperimentalFeature PSCommandNotFoundSuggestion
```

Output

WARNING: Enabling and disabling experimental features do not take effect until next start of PowerShell.

When an experimental feature becomes *mainstream*, it's no longer available as an experimental feature because the functionality is now part of the PowerShell engine or module. For example, the `PSAnsiRenderingFileInfo` feature became mainstream in PowerShell 7.3. You get the functionality of the feature automatically.

Note

Some features have configuration requirements, such as preference variables, that must be set to get the desired results from the feature.

When an experimental feature is *discontinued*, that feature is no longer available in the PowerShell. For example, the `PSNativePSPPathResolution` feature was discontinued in PowerShell 7.3.

Available features

This article describes the experimental features that are available and how to use the feature.

Legend

- The  icon indicates that the experimental feature is available in the version of PowerShell
- The  icon indicates the version of PowerShell where the experimental feature became mainstream
- The  icon indicates the version of PowerShell where the experimental feature was removed

[Expand table](#)

Name	7.4	7.5	7.6 (preview)
PSCommandNotFoundSuggestion			
PSDesiredStateConfiguration.InvokeDscResource			
PSSubsystemPluginModel			
PSLoadAssemblyFromNativeCode			
PSFeedbackProvider			
PSModuleAutoLoadSkipOfflineFiles			
PSCommandWithArgs			
PSNativeWindowsTildeExpansion			
PSRedirectToVariable			
PSSerializeJSONLongEnumAsNumber			

PSCommandNotFoundSuggestion

Note

This feature became mainstream in PowerShell 7.5-preview.5.

Recommends potential commands based on fuzzy matching search after a [CommandNotFoundException](#).

```
PS> get
```

Output

```
get: The term 'get' isn't recognized as the name of a cmdlet, function,
script file,
or operable program. Check the spelling of the name, or if a path was
included, verify
that the path is correct and try again.
```

```
Suggestion [4,General]: The most similar commands are: set, del, ft, gal,
gbp, gc, gci,
gcm, gdr, gcs.
```

PSCommandWithArgs

ⓘ Note

This feature became mainstream in PowerShell 7.5-preview.5.

This feature enables the `-CommandWithArgs` parameter for `pwsh`. This parameter allows you to execute a PowerShell command with arguments. Unlike `-Command`, this parameter populates the `$args` built-in variable that can be used by the command.

The first string is the command and subsequent strings delimited by whitespace are the arguments.

For example:

PowerShell

```
pwsh -CommandWithArgs '$args | % { "arg: $_" }' arg1 arg2
```

This example produces the following output:

Output

```
arg: arg1
arg: arg2
```

This feature was added in PowerShell 7.4-preview.2.

PSDesiredStateConfiguration.InvokeDscResource

Enables compilation to MOF on non-Windows systems and enables the use of `Invoke-DscResource` without an LCM.

Beginning with PowerShell 7.2, the **PSDesiredStateConfiguration** module was removed and this feature is disabled by default. To enable this feature you must install the **PSDesiredStateConfiguration** v2.0.5 module from the PowerShell Gallery and enable the feature.

DSC v3 doesn't have this experimental feature. DSC v3 only supports `Invoke-DscResource` and doesn't use or support MOF compilation. For more information, see [PowerShell Desired State Configuration v3](#).

PSFeedbackProvider

When you enable this feature, PowerShell uses a new feedback provider to give you feedback when a command can't be found. The feedback provider is extensible, and can be implemented by third-party modules. The feedback provider can be used by other subsystems, such as the predictor subsystem, to provide predictive IntelliSense results.

This feature includes two built-in feedback providers:

- **GeneralCommandErrorFeedback** serves the same suggestion functionality existing today
- **UnixCommandNotFound**, available on Linux, provides feedback similar to bash.

The **UnixCommandNotFound** serves as both a feedback provider and a predictor. The suggestion from command-not-found command is used both for providing the feedback when command can't be found in an interactive run, and for providing predictive IntelliSense results for the next command line.

This feature was added in PowerShell 7.4-preview.3.

PSLoadAssemblyFromNativeCode

Exposes an API to allow assembly loading from native code.

PSModuleAutoLoadSkipOfflineFiles

Note

This feature became mainstream in PowerShell 7.5-preview.5.

With this feature enabled, if a user's **PSModulePath** contains a folder from a cloud provider, such as OneDrive, PowerShell no longer triggers the download of all files contained within that folder. Any file marked as not downloaded are skipped. Users who use cloud providers to sync their modules between machines should mark the module folder as **Pinned** or the equivalent status for providers other than OneDrive. Marking the module folder as **Pinned** ensures that the files are always kept on disk.

This feature was added in PowerShell 7.4-preview.1.

PSRedirectToVariable

ⓘ Note

This experimental feature was added in PowerShell 7.5-preview.4.

When enabled, this feature adds support for redirecting to the Variable: drive. This feature allows you to redirect data to a variable using the `Variable:name` syntax. PowerShell inspects the target of the redirection and if it uses the Variable provider it calls `Set-Variable` rather than `Out-File`.

The following example shows how to redirect the output of a command to a Variable:

PowerShell

```
. {
    "Output 1"
    Write-Warning "Warning, Warning!"
    "Output 2"
} 3> Variable:warnings
$warnings
```

Output

```
Output 1
Output 2
WARNING: Warning, Warning!
```

PSSubsystemPluginModel

This feature enables the subsystem plugin model in PowerShell. The feature makes it possible to separate components of `System.Management.Automation.dll` into individual subsystems that reside in their own assembly. This separation reduces the disk footprint of the core PowerShell engine and allows these components to become optional features for a minimal PowerShell installation.

Currently, only the **CommandPredictor** subsystem is supported. This subsystem is used along with the `PSReadLine` module to provide custom prediction plugins. In future, **Job**, **CommandCompleter**, **Remoting** and other components could be separated into subsystem assemblies outside of `System.Management.Automation.dll`.

The experimental feature includes a new cmdlet, [Get-PSSubsystem](#). This cmdlet is only available when the feature is enabled. This cmdlet returns information about the subsystems that are available on the system.

PSNativeWindowsTildeExpansion

When this feature is enabled, PowerShell expands unquoted tilde (~) to the user's current home folder before invoking native commands. The following examples show how the feature works.

With the feature disabled, the tilde is passed to the native command as a literal string.

```
PowerShell  
PS> cmd.exe /c echo ~  
~
```

With the feature enabled, PowerShell expands the tilde before it's passed to the native command.

```
PowerShell  
PS> cmd.exe /c echo ~  
C:\Users\username
```

This feature only applies to Windows. On non-Windows platforms, tilde expansion is handled natively.

This feature was added in PowerShell 7.5-preview.2.

PSSerializeJSONLongEnumAsNumber

This feature enables the cmdlet [ConvertTo-Json](#) to serialize any enum values based on `Int64/long` or `UInt64/ulong` as a numeric value rather than the string representation of that enum value. This aligns the behavior of enum serialization with other enum base types where the cmdlet serializes enums as their numeric value. Use the `EnumsAsStrings` parameter to serialize as the string representation.

For example:

PowerShell

```
# PSSerializeJSONLongEnumAsNumber disabled
@{
    Key =
[System.Management.Automation.Tracing.PowerShellTraceKeywords]::Cmdlets
} | ConvertTo-Json
# { "Key": "Cmdlets" }

# PSSerializeJSONLongEnumAsNumber enabled
@{
    Key =
[System.Management.Automation.Tracing.PowerShellTraceKeywords]::Cmdlets
} | ConvertTo-Json
# { "Key": 32 }

# -EnumsAsStrings to revert back to the old behaviour
@{
    Key =
[System.Management.Automation.Tracing.PowerShellTraceKeywords]::Cmdlets
} | ConvertTo-Json -EnumsAsStrings
# { "Key": "Cmdlets" }
```

Using aliases

Article • 07/23/2024

An alias is an alternate name or shorthand name for a cmdlet or for a command element, such as a function, script, file, or executable file. You can run the command using the alias instead of the executable name.

Managing command aliases

PowerShell provides cmdlets for managing command aliases. The following command shows the cmdlets that manage aliases.

PowerShell

```
Get-Command -Noun Alias
```

Output

CommandType	Name	Version	Source
Cmdlet	Export-Alias	7.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	Get-Alias	7.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	Import-Alias	7.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	New-Alias	7.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	Remove-Alias	7.0.0.0	Microsoft.PowerShell.Utility
Cmdlet	Set-Alias	7.0.0.0	Microsoft.PowerShell.Utility

For more information, see [about_Aliases](#).

Use the [Get-Alias](#) cmdlet to list the aliases available in your environment. To list the aliases for a single cmdlet, use the **Definition** parameter and specify the executable name.

PowerShell

```
Get-Alias -Definition Get-ChildItem
```

Output

CommandType	Name
Alias	dir -> Get-ChildItem
Alias	gci -> Get-ChildItem
Alias	ls -> Get-ChildItem

To get the definition of a single alias, use the `Name` parameter.

PowerShell

```
Get-Alias -Name gci
```

Output

CommandType	Name
Alias	gci -> Get-ChildItem

To create an alias, use the `Set-Alias` command. You can create aliases for cmdlets, functions, scripts, and native executables files.

PowerShell

```
Set-Alias -Name np -Value Notepad.exe
Set-Alias -Name cmpo -Value Compare-Object
```

Compatibility aliases in Windows

PowerShell has several aliases that allow **Unix** and `cmd.exe` users to use familiar commands in Windows. The following table show common commands, the related PowerShell cmdlet, and the PowerShell alias:

[] Expand table

Windows Command Shell	Unix command	PowerShell cmdlet	PowerShell alias
<code>cd</code> , <code>chdir</code>	<code>cd</code>	<code>Set-Location</code>	<code>sl</code> , <code>cd</code> , <code>chdir</code>
<code>cls</code>	<code>clear</code>	<code>Clear-Host</code>	<code>cls</code> <code>clear</code>
<code>copy</code>	<code>cp</code>	<code>Copy-Item</code>	<code>cpi</code> , <code>cp</code> , <code>copy</code>
<code>del</code> , <code>erase</code> , <code>rd</code> , <code>rmdir</code>	<code>rm</code>	<code>Remove-Item</code>	<code>ri</code> , <code>del</code> , <code>erase</code> , <code>rd</code> , <code>rm</code> , <code>rmdir</code>
<code>dir</code>	<code>ls</code>	<code>Get-ChildItem</code>	<code>gci</code> , <code>dir</code> , <code>ls</code>
<code>echo</code>	<code>echo</code>	<code>Write-Output</code>	<code>write</code> <code>echo</code>
<code>md</code>	<code>mkdir</code>	<code>New-Item</code>	<code>ni</code>
<code>move</code>	<code>mv</code>	<code>Move-Item</code>	<code>mi</code> , <code>move</code> , <code>mi</code>

Windows Command Shell	Unix command	PowerShell cmdlet	PowerShell alias
popd	popd	Pop-Location	popd
	pwd	Get-Location	gl, pwd
pushd	pushd	Push-Location	pushd
ren	mv	Rename-Item	rni, ren
type	cat	Get-Content	gc, cat, type

! Note

The aliases in this table are Windows-specific. Some aliases aren't available on other platforms. This is to allow the native command to work in a PowerShell session. For example, `ls` isn't defined as a PowerShell alias on macOS or Linux so that the native command is run instead of `Get-ChildItem`.

Creating alternate names for commands with parameters

You can assign an alias to a cmdlet, script, function, or executable file. Unlike some Unix shells, you cannot assign an alias to a command with parameters. For example, you can assign an alias to the `Get-EventLog` cmdlet, but you cannot assign an alias to the `Get-EventLog -LogName System` command. You must create a function that contains the command with parameters.

For more information, see [about_Aliases](#).

Parameter aliases and shorthand names

PowerShell also provides ways to create shorthand names for parameters. Parameter aliases are defined using the `Alias` attribute when you declare the parameter. These can't be defined using the `*-Alias` cmdlets.

For more information, see the [Alias attribute](#) documentation.

In addition to parameter aliases, PowerShell lets you specify the parameter name using the fewest characters needed to uniquely identify the parameter. For example, the `Get-ChildItem` cmdlet has the `Recurse` and `ReadOnly` parameters. To uniquely identify the `Recurse` parameter

you only need to provide `-Rec`. If you combine that with the command alias, `Get-ChildItem -Recurse` can be shortened to `dir -Rec`.

Don't use aliases in scripts

Aliases are a convenience feature to be used interactively in the shell. You should always use the full command and parameter names in your scripts.

- Aliases can be deleted or redefined in a profile script
- Any aliases you define may not be available to the user of your scripts
- Aliases make your code harder to read and maintain

PowerShell learning resources

Article • 09/20/2022

Additional resources for learning about PowerShell.

Learn modules

Microsoft Learn is a free, online training platform that provides interactive learning for Microsoft products and more. Our goal is to help you become proficient on our technologies and learn more skills with fun, guided, hands-on, interactive content that's specific to your role and goals.

- [PowerShell modules](#)

Blogs and community

In addition to the Help available at the command line, the following resources provide more information for users who want to run PowerShell.

- [PowerShell Team Blog ↗](#). The best resource for learning directly from the PowerShell product team.
- [PowerShell Community Blog ↗](#) articles are scenario-driven. Written by the community, for the community.
- Have questions about using PowerShell? Connect with hundreds of other people who have similar interests in one of the many community forums listed on the [PowerShell Community](#) page.

Microsoft Virtual Academy

The Microsoft Virtual Academy videos have been moved to Channel 9.

- [Getting Started with Microsoft PowerShell](#)
- [Advanced Tools & Scripting with PowerShell 3.0 Jump Start](#)
- [Testing PowerShell with Pester](#)
- [Getting Started with PowerShell Desired State Configuration \(DSC\)](#)
- [Advanced PowerShell DSC and Custom Resources](#)
- [SharePoint Automation with DSC](#)

Resources for PowerShell Developers

The following resources provide resources to help developers create their own PowerShell modules, functions, cmdlets, providers, and hosting applications.

- [PowerShell SDK](#)
- [PowerShell SDK API Browser](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

PowerShell Glossary

Article • 04/10/2024

This article lists common terms used to talk about PowerShell.

B

binary module

A PowerShell module whose root module is a binary (`.dll`) file. A binary module may or may not include a module manifest.

C

CommonParameter

A parameter that's added to all cmdlets, advanced functions, and workflows by the PowerShell engine.

D

dot source

In PowerShell, to start a command by typing a dot and a space before the command. Commands that are dot sourced run in the current scope instead of in a new scope. Any variables, aliases, functions, or drives that command creates are created in the current scope and are available to users when the command is completed.

dynamic module

A module that exists only in memory. The `New-Module` and `Import-PSSession` cmdlets create dynamic modules.

dynamic parameter

A parameter that's added to a PowerShell cmdlet, function, or script under certain conditions. Cmdlets, functions, providers, and scripts can add dynamic parameters.

F

format file

A PowerShell XML file that has the `.format.ps1xml` extension and that defines how PowerShell displays an object based on its .NET Framework type.

G

global session state

The session state that contains the data that's accessible to the user of a PowerShell session.

H

Host

The interface that the PowerShell engine uses to communicate with the user. For example, the host specifies how prompts are handled between PowerShell and the user.

host application

A program that loads the PowerShell engine into its process and uses it to perform operations.

I

input processing method

A method that a cmdlet can use to process the records it receives as input. The input processing methods include the `BeginProcessing` method, the `ProcessRecord` method, the `EndProcessing` method, and the `StopProcessing` method.

M

manifest module

A PowerShell module that has a manifest and whose `RootModule` key is empty.

member-access enumeration

A PowerShell convenience feature to automatically enumerate items in a collection when using the member-access operator (`.`).

module

A self-contained reusable unit that allows you to partition, organize, and abstract your PowerShell code. A module can contain cmdlets, providers, functions, variables, and other types of resources that can be imported as a single unit.

module manifest

A PowerShell data file (`.psd1`) that describes the contents of a module and that controls how a module is processed.

module session state

The session state that contains the public and private data of a PowerShell module. The private data in this session state isn't available to the user of a PowerShell session.

N

non-terminating error

An error that doesn't stop PowerShell from continuing to process the command. See also, [terminating error](#).

noun

The word that follows the hyphen in a PowerShell cmdlet name. The noun describes the resources upon which the cmdlet acts.

P

parameter set

A group of parameters that can be used in the same command to perform a specific action.

pipe

In PowerShell, to send the results of the preceding command as input to the next command in the pipeline.

pipeline

A series of commands connected by pipeline operators (`|`). Each pipeline operator sends the results of the preceding command as input to the next command.

PowerShell cmdlet

A single command that participates in the pipeline semantics of PowerShell. This includes binary (C#) cmdlets, advanced script functions, CDXML, and Workflows.

PowerShell command

The elements in a pipeline that cause an action to be carried out. PowerShell commands are either typed at the keyboard or invoked programmatically.

PowerShell data file

A text file that has the `.psd1` file extension. PowerShell uses data files for various purposes such as storing module manifest data and storing translated strings for script internationalization.

PowerShell drive

A virtual drive that provides direct access to a data store. It can be defined by a PowerShell provider or created at the command line. Drives created at the command line are session-specific drives and are lost when the session is closed.

provider

A Microsoft .NET Framework-based program that makes the data in a specialized data store available in PowerShell so that you can view and manage it.

PSSession

A type of PowerShell session that's created, managed, and closed by the user.

R

root module

The module specified in the **RootModule** key in a module manifest.

runspace

In PowerShell, the operating environment in which each command in a pipeline is executed.

S

scalar value

In PowerShell, a scalar value is any value type that is not enumerable. This includes the .NET primitive types, such as booleans and numbers, and other value types such as **String**, **DateTime** and **Guid**.

For a list of .NET primitive types, see the *Remarks* section of [System.Type.IsPrimitive Property](#).

script block

In the PowerShell programming language, a collection of statements or expressions that can be used as a single unit. A script block can accept arguments and return values.

script file

A file that has the `.ps1` extension and contains a script written in the PowerShell language.

script module

A PowerShell module whose root module is a script module (`.psm1`) file. A script module may include a module manifest. The script defines the members that the script module exports.

shell

The command interpreter that's used to pass commands to the operating system.

switch parameter

A parameter that doesn't take an argument. The value of a switch parameter defaults to `$false`. When a switch parameter is used, its value becomes `$true`.

T

terminating error

An error that stops PowerShell from processing the command. See also, [non-terminating error](#).

transaction

An atomic unit of work. The work in a transaction must be completed as a whole. If any part of the transaction fails, the entire transaction fails.

type file

A PowerShell XML file that has the `.types.ps1xml` extension and that extends the properties of Microsoft .NET Framework types in PowerShell.

V

verb

The word that precedes the hyphen in a PowerShell cmdlet name. The verb describes the action that the cmdlet performs.

W

Windows PowerShell ISE

The Integrated Scripting Environment (ISE) - A Windows PowerShell host application that enables you to run commands and to write, test, and debug scripts in a friendly, syntax-colored, Unicode-compliant environment.

Windows PowerShell snap-in

A resource that defines a set of cmdlets, providers, and Microsoft .NET Framework types that can be added to the Windows PowerShell environment. PowerShell snap-ins have been replaced by modules.

Windows PowerShell Workflow

A workflow is a sequence of programmed, connected steps that perform long-running tasks or require the coordination of multiple steps across multiple devices or managed nodes. Windows PowerShell Workflow lets IT pros and developers author sequences of multi-device management activities, or single tasks within a workflow, as workflows. Windows PowerShell Workflow lets you adapt and run both PowerShell scripts and XAML files as workflows. Windows PowerShell Workflow is built on the Windows Workflow Foundation, which has been deprecated.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Overview of what's new in PowerShell

A collection of release notes and documentation about the new features available in new versions of PowerShell.

What's new in PowerShell 7

WHAT'S NEW

[What's new in PowerShell 7.6 \(preview\)](#)

[What's new in PowerShell 7.5 \(RC\)](#)

[What's new in PowerShell 7.4 \(LTS\)](#)

[Differences between Windows PowerShell 5.1 and PowerShell 7](#)

[PowerShell differences on non-Windows platforms](#)

What's new in PowerShell 5.1

WHAT'S NEW

[What is Windows PowerShell?](#)

[Differences between Windows PowerShell 5.1 and PowerShell 7](#)

[Migrating from Windows PowerShell 5.1 to PowerShell 7](#)

History & Compatibility

WHAT'S NEW

[Release history of modules and cmdlets](#)

[Module compatibility](#)

[Previous versions of PowerShell](#)

What's New in PowerShell 7.6

Article • 04/09/2025

PowerShell 7.6-preview.4 includes the following features, updates, and breaking changes. PowerShell 7.6 is built on .NET 9.0.101 GA release.

For a complete list of changes, see the [CHANGELOG](#) in the GitHub repository.

Updated modules

PowerShell 7.6-preview.4 includes the following updated modules:

- `Microsoft.PowerShell.PSResourceGet` v1.1.0
- `PSReadLine` v2.3.6
- `Microsoft.PowerShell.ThreadJob` v2.2.0
- `ThreadJob` v2.1.0

The `ThreadJob` was renamed to `Microsoft.PowerShell.ThreadJob` module. There is no difference in the functionality of the module. To ensure backward compatibility for scripts that use the old name, the `ThreadJob` v2.1.0 module is a proxy module that points to the `Microsoft.PowerShell.ThreadJob` v2.2.0.

Breaking Changes

- Fix `WildcardPattern.Escape` to escape lone backticks correctly ([#25211](#)) (Thanks @ArmaanMcleod!)
- Convert `-ChildPath` parameter to `string[]` for `Join-Path` cmdlet ([#24677](#)) (Thanks @ArmaanMcleod!)
- Remove trailing space from event source name ([#24192](#)) (Thanks @MartinGC94!)

Tab completion improvements

- Update Named and Statement block type inference to not consider `AssignmentStatements` and Increment/decrement operators as part of their output ([#21137](#)) (Thanks @MartinGC94!)
- Add `-PropertyType` argument completer for `New-ItemProperty` ([#21117](#)) (Thanks @ArmaanMcleod!)
- Add completion single/double quote support for `-Noun` parameter for `Get-Command` ([#24977](#)) (Thanks @ArmaanMcleod!)

- Add completion single/double quote support for `-PSEdition` parameter for `Get-Module` (#24971) (Thanks @ArmaanMcleod!)
- Convert `InvalidCommandNameCharacters` in `AnalysisCache` to `SearchValues<char>` for more efficient char searching (#24880) (Thanks @ArmaanMcleod!)
- Convert `s_charactersRequiringQuotes` in `Completion Completers` to `SearchValues<char>` for more efficient char searching (#24879) (Thanks @ArmaanMcleod!)
- Update `IndexOfAny()` calls with invalid path/filename to `SearchValues<char>` for more efficient char searching ([#24896][24896]) (Thanks @ArmaanMcleod!)
- Replace `char[]` array in `CompletionRequiresQuotes` with cached `SearchValues<char>` (#24907) (Thanks @ArmaanMcleod!)
- Add quote handling in `Verb`, `StrictModeVersion`, `Scope` and `PropertyType` Argument Completers with single helper method (#24839) (Thanks @ArmaanMcleod!)
- Fix share completion with provider and spaces (#19440) (Thanks @MartinGC94!)
- Improve variable type inference (#19830) (Thanks @MartinGC94!)
- Add tooltips for hashtable key completions (#17864) (Thanks @MartinGC94!)
- Fix type inference of parameters in classic functions (#25172) (Thanks @MartinGC94!)
- Improve assignment type inference (#21143) (Thanks @MartinGC94!)
- Exclude `OutVariable` assignments within the same `CommandAst` when inferring variables (#25224) (Thanks @MartinGC94!)
- Fix parameter completion when script requirements fail (#17687) (Thanks @MartinGC94!)
- Improve the completion for attribute arguments (#25129) (Thanks @MartinGC94!)
- Fix completion that relies on pseudobinding in script blocks (#25122) (Thanks @MartinGC94!)
- Don't complete duplicate command names (#21113) (Thanks @MartinGC94!)
- Add completion for variables assigned by command redirection (#25104) (Thanks @MartinGC94!)
- Fix `TypeName.GetReflectionType()` to work when the `TypeName` instance represents a generic type definition within a `GenericTypeDefinition` (#24985)
- Update variable/property assignment completion so it can fallback to type inference (#21134) (Thanks @MartinGC94!)
- Handle type inference for redirected commands (#21131) (Thanks @MartinGC94!)
- Use `Get-Help` approach to find `about_*.help.txt` files with correct locale for completions (#24194) (Thanks @MartinGC94!)
- Fix completion of variables assigned inside Do loops (#25076) (Thanks @MartinGC94!)
- Fix completion of provider paths when a path returns itself instead of its children (#24755) (Thanks @MartinGC94!)
- Enable completion of scoped variables without specifying scope (#20340) (Thanks @MartinGC94!)

- Fix issue with incomplete results when completing paths with wildcards in non-filesystem providers ([#24757](#)) (Thanks @MartinGC94!)

Cmdlet improvements

- Add `-ExcludeModule` parameter to `Get-Command` ([#18955](#)) (Thanks @MartinGC94!)
- Return correct `FileName` property for `Get-Item` when listing alternate data streams ([#18019](#)) (Thanks @kilasuit!)
- Fix `Get-ItemProperty` to report non-terminating error for cast exception ([#21115](#)) (Thanks @ArmaanMcleod!)
- Fix a bug in how q handles XmlNode object ([#24669](#)) (Thanks @brendandburns!)
- Error when `New-Item -Force` is passed an invalid directory name ([#24936](#)) (Thanks @kborowinski!)
- Allow `Start-Transcript` to use `$Transcript` which is a `PSObject` wrapped string to specify the transcript path ([#24963](#)) (Thanks @kborowinski!)
- Improve `Start-Process -Wait` polling efficiency ([#24711](#)) (Thanks @jborean93!)
- Add completion of modules by their shortname ([#20330](#)) (Thanks @MartinGC94!)

Engine improvements

- Added the AIShell module to telemetry collection list ([#24747](#))
- Added helper in `EnumSingleTypeConverter` to get enum names as array ([#17785](#)) (Thanks @fflaten!)
- Update `DnsNameList` for `X509Certificate2` to use `X509SubjectAlternativeNameExtension.EnumerateDnsNames()` Method ([#24714](#)) (Thanks @ArmaanMcleod!)
- Stringify `ErrorRecord` with empty exception message to empty string ([#24949](#)) (Thanks @MatejKafka!)
- Add `PipelineStopToken` to `Cmdlet` which will be signaled when the pipeline is stopping ([#24620](#)) (Thanks @jborean93!)
- Fallback to AppLocker after `W1dpCanExecuteFile` ([#24912](#))
- Move .NET method invocation logging to after the needed type conversion is done for method arguments ([#25022](#))
- Fix infinite loop in variable type inference ([#25206](#)) (Thanks @MartinGC94!)
- Remove the old fuzzy suggestion and fix the local script file name suggestion ([#25177](#))
- Make `SystemPolicy` public APIs visible but non-op on Unix platforms so that they can be included in `PowerShellStandard.Library` ([#25051](#))
- Set standard handles explicitly when starting a process with `-NoNewWindow` ([#25061](#))
- Fix tooltip for variable expansion and include desc ([#25112](#)) (Thanks @jborean93!)

- Allow empty prefix string in 'Import-Module -Prefix' to override default prefix in manifest (#20409) (Thanks @MartinGC94!)
- Use script filepath when completing relative paths for using statements (#20017) (Thanks @MartinGC94!)
- Allow DSC parsing through OS architecture translation layers (#24852) (Thanks @bdeb1337!)

Experimental features

The following experimental features are included in PowerShell 7.6-preview.3:

- `PSNativeWindowsTildeExpansion` - Add tilde expansion for Windows-native executables
- `PSRedirectToVariable` - Allow redirecting to a variable
- `PSSerializeJSONLongEnumAsNumber` - `ConvertTo-Json` now treats large enums as numbers

What's New in PowerShell 7.5

Article • 04/25/2025

PowerShell 7.5.1 includes the following features, updates, and breaking changes. PowerShell 7.5 is built on .NET 9.0.203 release.

For a complete list of changes, see the [CHANGELOG](#) in the GitHub repository. For more information about .NET 9, see [What's new in .NET 9][07].

Breaking Changes

- Fix `-OlderThan` and `-NewerThan` parameters for `Test-Path` when using `PathType` and date range ([#20942](#)) (Thanks @ArmaanMcleod!)
 - Previously `-OlderThan` would be ignored if specified together
- Change `New-FileCatalog -CatalogVersion` default to 2 ([#20428](#)) (Thanks @ThomasNieto!)
- Block getting help from network locations in restricted remoting sessions ([#20593](#))
- The Windows installer now remembers installation options used and uses them to initialize options for the next installation ([#20420](#)) (Thanks @reduckted!)
- `ConvertTo-Json` now serializes `BigInteger` as a number ([#21000](#)) (Thanks @jborean93!)
- .NET 9 removed the `BinaryFormatter` implementation causing a regression in the `Out-GridView` cmdlet. The search feature of `Out-GridView` doesn't work in PowerShell 7.5. This problem is tracked in [Issue #24749](#).

Updated modules

PowerShell 7.5.0 includes the following updated modules:

- `Microsoft.PowerShell.PSResourceGet` v1.1.0
- `PSReadLine` v2.3.6

Tab completion improvements

Many thanks to @ArmaanMcleod and others for all their work to improve tab completion.

- Fall back to type inference when hashtable key-value cannot be retrieved from safe expression ([#21184](#)) (Thanks @MartinGC94!)
- Fix the regression when doing type inference for `$_` ([#21223](#)) (Thanks @MartinGC94!)
- Expand `~` to `$HOME` on Windows with tab completion ([#21529](#))

- Don't complete when declaring parameter name and class member (#21182 ↗) (Thanks @MartinGC94!)
- Prevent fallback to file completion when tab completing type names (#20084 ↗) (Thanks @MartinGC94)
- Add argument completer to `-Version` for `Set-StrictMode` (#20554 ↗) (Thanks @ArmaanMcleod!)
- Add `-Verb` argument completer for `Get-Verb` / `Get-Command` and refactor `Get-Verb` (#20286 ↗) (Thanks @ArmaanMcleod)
- Add `-Verb` argument completer for `Start-Process` (#20415 ↗) (Thanks @ArmaanMcleod)
- Add `-Scope` argument completer for `*-Variable`, `*-Alias` & `*-PSDrive` commands (#20451 ↗) (Thanks @ArmaanMcleod)
- Add `-Module` completion for `Save-Help`/`Update-Help` commands (#20678 ↗) (Thanks @ArmaanMcleod)

New cmdlets

- Add `ConvertTo-CliXml` and `ConvertFrom-CliXml` cmdlets (#21063 ↗) (Thanks @ArmaanMcleod!)

Web cmdlets improvements

- Fix to allow `-PassThru` and `-Outfile` work together (#24086 ↗)
- Add `OutFile` property in `WebResponseObject` (#24047 ↗)
- Show filename in `Invoke-WebRequest -OutFile -Verbose` (#24041 ↗)
- Fix WebCmdlets when `-Body` is specified but `ContentType` is not (#23952 ↗) (Thanks @CarloToso!)
- Fix `Invoke-WebRequest` to report correct size when `-Resume` is specified (#20207 ↗) (Thanks @LNKLEO!)
- Fix Web Cmdlets to allow `WinForm` apps to work correctly (#20606 ↗)

Other cmdlet improvements

- Enable `-NoRestart` to work with `Register-PSSessionConfiguration` (#23891 ↗)
- Add `IgnoreComments` and `AllowTrailingCommas` options to `Test-Json` cmdlet (#23817 ↗) (Thanks @ArmaanMcleod!)
- Get-Help may report parameters with `ValueFromRemainingArguments` attribute as pipeline-able (#23871 ↗)

- Change type of `LineNumber` to `ulong` in `Select-String` (#24075) (Thanks @Snowman-25!)
- `Get-Process`: Remove admin requirement for `-IncludeUserName` (#21302) (Thanks @jborean93!)
- Fix `Test-Path -IsValid` to check for invalid path and filename characters (#21358)
- Add `RecommendedAction` to `ConciseView` of the error reporting (#20826) (Thanks @JustinGrote!)
- Added progress bar for `Remove-Item` cmdlet (#20778) (Thanks @ArmaanMcleod!)
- Fix `Test-Connection` due to .NET 8 changes (#20369)
- Fix `Get-Service` non-terminating error message to include category (#20276)
- Add `-Empty` and `-InputObject` parameters to `New-Guid` (#20014) (Thanks @CarloToso!)
- Add the alias `r` to the parameter `-Recurse` for the `Get-ChildItem` command (#20100) (Thanks @kilasuit!)
- Add `LP` to `LiteralPath` aliases for functions still missing it (#20820)
- Add implicit localization fallback to `Import-LocalizedData` (#19896) (Thanks @chrisdent-de!)
- Add `Aliases` to the properties shown up when formatting the help content of the parameter returned by `Get-Help` (#20994)
- Add `HelpUri` to `Remove-Service` (#20476)
- Fix completion crash for the SCCM provider (#20815, #20919, #20915) (Thanks @MartinGC94!)
- Fix regression in `Get-Content` when `-Tail 0` and `-Wait` are used together (#20734) (Thanks @CarloToso!)
- Fix `Start-Process -PassThru` to make sure the `ExitCode` property is accessible for the returned `Process` object (#20749) (Thanks @CodeCyclone!)
- Fix `Group-Object` to use current culture for its output (#20608)
- Fix `Group-Object` output using interpolated strings (#20745) (Thanks @mawosoft!)
- Fix rendering of `DisplayRoot` for network `PSDrive` (#20793)
- Fix `Copy-Item` progress to only show completed when all files are copied (#20517)
- Fix UNC path completion regression (#20419) (Thanks @MartinGC94!)
- Report error if invalid `-ExecutionPolicy` is passed to `pwsh` (#20460)
- Add `WinGetCommandNotFound` and `CompletionPredictor` modules to track usage (#21040)
- Add `DateKind` parameter to `ConvertFrom-Json` (#20925) (Thanks @jborean93!)
- Add `DirectoryInfo` to the `OutputType` for `New-Item` (#21126) (Thanks @MartinGC94!)
- Fix `Get-Error` serialization of array values (#21085) (Thanks @jborean93!)
- Fix `Test-ModuleManifest` so it can use a UNC path (#24115)

- Fix `Get-TypeData` to write to the pipeline immediately instead of collecting data first (#24236) (Thanks @MartinGC94)
- Add `-Force` parameter to `Resolve-Path` and `Convert-Path` cmdlets to support wildcard hidden files #20981 (Thanks @ArmaanMcleod!)

Engine improvements

- Fallback to AppLocker after `WldpCanExecuteFile` (#25305)
- Explicitly start and stop ANSI Error Color (#24065) (Thanks @JustinGrote!)
- Improve .NET overload definition of generic methods (#21326) (Thanks @jborean93!)
- Optimize the `+=` operation for a collection when it's an object array (#23901) (Thanks @jborean93!)
- Add telemetry to check for specific tags when importing a module (#20371)
- Add `PSAdapter` and `ConsoleGuiTools` to module load telemetry allowlist (#20641)
- Add WinGet module to track usage (#21040)
- Ensure the filename is not null when logging WDAC ETW events (#20910) (Thanks @jborean93!)
- Fix four regressions introduced by the WDAC logging feature (#20913)
- Leave the input, output, and error handles unset when they are not redirected (#20853)
- Fix implicit remoting proxy cmdlets to act on common parameters (#20367)
- Include the module version in error messages when module is not found (#20144) (Thanks @ArmaanMcleod!)
- Fix `unixmode` to handle `setuid` and `sticky` when file is not an executable (#20366)
- Fix using assembly to use `Path.Combine` when constructing assembly paths (#21169)
- Validate the value for using namespace during semantic checks to prevent declaring invalid namespaces (#21162)
- Handle global tool specially when prepending `$PSHOME` to PATH (#24228)

Experimental features

The following experimental features were converted to mainstream features in PowerShell 7.5-rc.1:

- `PSCommandNotFoundSuggestion`
- `PSCommandWithArgs`
- `PSModuleAutoLoadSkipOfflineFiles`

The following experimental features are included in PowerShell 7.5-rc.1:

- `PSRedirectToVariable` - Allow redirecting to a variable (#20381)

- [PSNativeWindowsTildeExpansion](#) - Add tilde expansion for Windows-native executables (#20402) (Thanks @domsleee!)
- [PSSerializeJSONLongEnumAsNumber](#) - `ConvertTo-Json` now treats large enums as numbers (#20999) (Thanks @jborean93!)

Performance improvements

PowerShell 7.5-rc.1 included [PR#23901](#) from @jborean93 that improves the performance of the `+ =` operation for an array of objects.

The following example measures the performance for different methods of adding elements to an array.

PowerShell

```
$tests = @{
    'Direct Assignment' = {
        param($count)

        $result = foreach($i in 1..$count) {
            $i
        }
    }
    'List<T>.Add(T)' = {
        param($count)

        $result = [Collections.Generic.List[int]]::new()
        foreach($i in 1..$count) {
            $result.Add($i)
        }
    }
    'Array+= Operator' = {
        param($count)

        $result = @()
        foreach($i in 1..$count) {
            $result += $i
        }
    }
}

5kb, 10kb | ForEach-Object {
    $groupResult = foreach($test in $tests.GetEnumerator()) {
        $ms = (Measure-Command { & $test.Value -Count $_ }).TotalMilliseconds

        [pscustomobject]@{
            CollectionSize      = $_
            Test                = $test.Key
            TotalMilliseconds = [Math]::Round($ms, 2)
        }
    }
}
```

```

[GC]::Collect()
[GC]::WaitForPendingFinalizers()
}

$groupResult = $groupResult | Sort-Object TotalMilliseconds
$groupResult | Select-Object *, @{
    Name      = 'RelativeSpeed'
    Expression = {
        $relativeSpeed = $_.TotalMilliseconds /
$groupResult[0].TotalMilliseconds
        $speed = [Math]::Round($relativeSpeed, 2).ToString() + 'x'
        if ($speed -eq '1x') { $speed } else { $speed + ' slower' }
    }
} | Format-Table -AutoSize
}

```

When you run the script in PowerShell 7.4.6, you see that using the `+EQ` operator is the slowest method.

Output

CollectionSize Test	TotalMilliseconds	RelativeSpeed
5120 Direct Assignment	4.17	1x
5120 List<T>.Add(T)	90.79	21.77x slower
5120 Array+= Operator	342.58	82.15x slower
CollectionSize Test	TotalMilliseconds	RelativeSpeed
10240 Direct Assignment	0.64	1x
10240 List<T>.Add(T)	184.10	287.66x slower
10240 Array+= Operator	1668.13	2606.45x slower

When you run the script in PowerShell 7.5-rc.1, you see that using the `+EQ` operator is much faster than PowerShell 7.4.6. Now, it's also faster than using the `List<T>.Add(T)` method.

Output

CollectionSize Test	TotalMilliseconds	RelativeSpeed
5120 Direct Assignment	4.71	1x
5120 Array+= Operator	40.42	8.58x slower
5120 List<T>.Add(T)	92.17	19.57x slower
CollectionSize Test	TotalMilliseconds	RelativeSpeed
10240 Direct Assignment	1.76	1x

10240 Array+= Operator	104.73 59.51x slower
10240 List<T>.Add(T)	173.00 98.3x slower

[07]: /dotnet/core/whats-new/dotnet-9/overview)

What's New in PowerShell 7.4

Article • 05/13/2025

PowerShell 7.4.10 includes the following features, updates, and breaking changes. PowerShell 7.4.10 is built on .NET 8.0.408.

For a complete list of changes, see the [CHANGELOG](#) in the GitHub repository.

Breaking changes

- Nano server docker images aren't available for this release
- Added the **ProgressAction** parameter to the Common Parameters
- Update some PowerShell APIs to throw **ArgumentException** instead of **ArgumentNullException** when the argument is an empty string ([#19215](#)) (Thanks @xtqqczze!)
- Remove code related to `#Requires -PSSnapin` ([#19320](#))
- `Test-Json` now uses JsonSchema.NET instead of Newtonsoft.Json.Schema.
 - With this change, `Test-Json` no longer supports the older Draft 4 schemas. ([#18141](#)) (Thanks @gregsdennis!). For more information about JSON schemas, see [JSON Schema](#) documentation. This also breaks `Test-Json` for JSON and JSONC files with comments.
 - `ConvertFrom-Json` support still uses Newtonsoft.Json.Schema so it can convert JSON files with comments.
- Output from `Test-Connection` now includes more detailed information about TCP connection tests
- .NET introduced changes that affected `Test-Connection`. The cmdlet now returns an error about the need to use `sudo` on Linux platforms when using a custom buffer size ([#20369](#))
- Experimental feature **PSNativeCommandPreserveBytePipe** is now mainstream. PowerShell now preserves the byte-stream data when redirecting the **stdout** stream of a native command to a file or when piping byte-stream data to the **stdin** stream of a native command.
- Change how relative paths in `Resolve-Path` are handled when using the **RelativebasePath** parameter ([#19755](#)) (Thanks @MartinGC94!)
- Remove unused PSv2 code - removes TabExpansion function ([#18337](#))

Installer updates

The Windows MSI package now provides an option to disable PowerShell telemetry during installation. For more information, see [Install the msi package from the command line](#).

Updated versions of PSResourceGet and PSReadLine

PowerShell 7.4 includes `Microsoft.PowerShell.PSResourceGet` v1.0.1. This module is installed side-by-side with `PowerShellGet` v2.2.5 and `PackageManagement` v1.4.8.1. For more information, see the documentation for [Microsoft.PowerShell.PSResourceGet](#).

PowerShell 7.4 now includes `PSReadLine` v2.3.4. For more information, see the documentation for [PSReadLine](#).

Tab completion improvements

Many thanks to [@MartinGC94](#) and others for all their work to improve tab completion.

- Fix issue when completing the first command in a script with an empty array expression ([#18355](#))
- Fix positional argument completion ([#17796](#))
- Prioritize the default parameter set when completing positional arguments ([#18755](#))
- Improve pseudo binding for dynamic parameters ([#18030](#))
- Improve type inference of hashtable keys ([#17907](#))
- Fix type inference error for empty return statements ([#18351](#))
- Improve type inference for Get-Random ([#18972](#))
- Fix type inference for all scope variables ([#18758](#))
- Improve enumeration of inferred types in pipeline ([#17799](#))
- Add completion for values in comparisons when comparing Enums ([#17654](#))
- Add property assignment completion for enums ([#19178](#))
- Fix completion for PSCustomObject variable properties ([#18682](#))
- Fix member completion in attribute argument ([#17902](#))
- Exclude redundant parameter aliases from completion results ([#19382](#))
- Fix class member completion for classes with base types ([#19179](#))
- Add completion for the `using` keyword ([#16514](#))
- Fix TabExpansion2 variable leak when completing variables ([#18763](#))
- Enable completion of variables across ScriptBlock scopes ([#19819](#))
- Fix completion of the foreach statement variable ([#19814](#))
- Fix variable type inference precedence ([#18691](#))
- Fix member completion for PowerShell Enum class ([#19740](#))
- Fix parsing for array literals in index expressions in method calls ([#19224](#))

- Improve path completion ([#19489 ↗](#))
- Fix an indexing out of bound error in `CompleteInput` for empty script input ([#19501 ↗](#))
- Improve variable completion performance ([#19595 ↗](#))
- Improve Hashtable key completion for type constrained variable assignments, nested Hashtables and more ([#17660 ↗](#))
- Infer external application output as strings ([#19193 ↗](#))
- Update parameter completion for enums to exclude values not allowed by `ValidateRange` attributes ([#17750 ↗](#)) (Thanks @fflaten!).
- Fix dynamic parameter completion ([#19510 ↗](#))
- Add completion for variables assigned by the `data` statement ([#19831 ↗](#))
- Fix expanding tilde (~) on Windows systems to `$HOME` to prevent breaking use cases with native commands ([#21529 ↗](#))

Web cmdlet improvements

Many thanks to [@CarloToso](#) and others for all the work on improving web cmdlets.

- Fix decompression in web cmdlets to include Brotli ([#17955 ↗](#)) (Thanks @iSazonov!)
- Webcmdlets add 308 to redirect codes and small cleanup ([#18536 ↗](#))
- Complete the progress bar rendering in `Invoke-WebRequest` when downloading is complete or cancelled ([#18130 ↗](#))
- Web cmdlets get **Retry-After** interval from response headers if the status code is 429 ([#18717 ↗](#))
- Web cmdlets set default charset encoding to UTF8 ([#18219 ↗](#))
- Preserve `WebSession.MaximumRedirection` from changes ([#19190 ↗](#))
- WebCmdlets parse XML declaration to get encoding value, if present. ([#18748 ↗](#))
- Fix using `xml -Body` in webcmdlets without an encoding ([#19281 ↗](#))
- Adjust PUT method behavior to POST one for default content type in WebCmdlets ([#19152 ↗](#))
- Take into account `ContentType` from Headers in WebCmdlets ([#19227 ↗](#))
- Allow to preserve the original HTTP method by adding `-PreserveHttpMethodOnRedirect` to Web cmdlets ([#18894 ↗](#))
- Webcmdlets display an error on https to http redirect ([#18595 ↗](#))
- Add **AllowInsecureRedirect** switch to Web cmdlets ([#18546 ↗](#))
- Improve verbose message in web cmdlets when content length is unknown ([#19252 ↗](#))
- Build the relative URI for links from the response in `Invoke-WebRequest` ([#19092 ↗](#))
- Fix redirection for `-CustomMethod POST` in WebCmdlets ([#19111 ↗](#))
- Dispose previous response in Webcmdlets ([#19117 ↗](#))
- Improve `Invoke-WebRequest` xml and json errors format ([#18837 ↗](#))

- Add `ValidateNotNullOrEmpty` to `OutFile` and `InFile` parameters of WebCmdlets (#19044 ↗)
- `HttpKnownHeaderNames` update headers list (#18947 ↗)
- `Invoke-RestMethod -FollowRelLink` fix links containing commas (#18829 ↗)
- Fix bug with managing redirection and `KeepAuthorization` in Web cmdlets (#18902 ↗)
- Add `StatusCode` to `HttpResponseException` (#18842 ↗)
- Support HTTP persistent connections in Web Cmdlets (#19249 ↗) (Thanks @stevenebutler!)
- Small cleanup `Invoke-RestMethod` (#19490 ↗)
- Improve the verbose message of WebCmdlets to show correct HTTP version (#19616 ↗)
- Add `FileNameStar` to `MultipartFileContent` in WebCmdlets (#19467 ↗)
- Fix HTTP status from 409 to 429 for WebCmdlets to get retry interval from `Retry-After` header. (#19622 ↗) (Thanks @mkht!)
- Change `-TimeoutSec` to `-ConnectionTimeoutSeconds` and add `-OperationTimeoutSeconds` to web cmdlets (#19558 ↗) (Thanks @stevenebutler!) Other cmdlets
- Support Ctrl+c when connection hangs while reading data in WebCmdlets (#19330 ↗) (Thanks @stevenebutler!)
- Support Unix domain socket in WebCmdlets (#19343 ↗)

Other cmdlet improvements

- `Test-Connection` now returns error about the need to use `sudo` on Linux platforms when using a custom buffer size (#20369 ↗)
- Add output types to Format commands (#18746 ↗) (Thanks @MartinGC94!)
- Add output type attributes for `Get-WinEvent` (#17948 ↗) (Thanks @MartinGC94!)
- Add `Path` and `LiteralPath` parameters to `Test-Json` cmdlet (#19042 ↗) (Thanks @ArmaanMcleod!)
- Add `NoHeader` parameter to `ConvertTo-Csv` and `Export-Csv` cmdlets (#19108 ↗) (Thanks @ArmaanMcleod!)
- Add `Confirm` and `WhatIf` parameters to `Stop-Transcript` (#18731 ↗) (Thanks @JohnLBevan!)
- Add `FuzzyMinimumDistance` parameter to `Get-Command` (#18261 ↗)
- Make `Encoding` parameter able to take `ANSI` encoding in PowerShell (#19298 ↗) (Thanks @CarloToso!)
- Add progress to `Copy-Item` (#18735 ↗)
- `Update-Help` now reports an error when using implicit culture on non-US systems. (#17780 ↗) (Thanks @dkaszews!)
- Don't require activity when creating a completed progress record (#18474 ↗) (Thanks @MartinGC94!)

- Disallow negative values for `Get-Content` cmdlet parameters `-Head` and `-Tail` (#19715 ↴) (Thanks @CarloToso!)
- Make `Update-Help` throw proper error when current culture isn't associated with a language (#19765 ↴) (Thanks @josea!)
- Allow combining of `-Skip` and `-SkipLast` parameters in `Select-Object` cmdlet. (#18849 ↴) (Thanks @ArmaanMcleod!)
- Add `Get-SecureRandom` cmdlet (#19587 ↴)
- `Set-Clipboard -AsOSC52` for remote usage (#18222 ↴) (Thanks @dkaszews!)
- Speed up `Resolve-Path` relative path resolution (#19171 ↴) (Thanks @MartinGC94!)
- Added the switch parameter `-CaseInsensitive` to `Select-Object` and `Get-Unique` cmdlets (#19683 ↴) (Thanks @ArmaanMcleod!)
- `Restart-Computer` and `Stop-Computer` should fail with error when not running via sudo on Unix (#19824 ↴)

Engine improvements

Updates to `$PSSStyle`

- Adds `Dim` and `DimOff` properties (#18653 ↴)
- Added static methods to the `PSSStyle` class that map foreground and background `ConsoleColor` values to ANSI escape sequences (#17938 ↴)
- Table headers for calculated fields are formatted in italics by default
- Add support of respecting `$PSSStyle.OutputRendering` on the remote host (#19601 ↴)
- Updated telemetry data to include use of `CrescendoBuilt` modules (#20371 ↴)

Other Engine updates

- Fallback to AppLocker after `WldpCanExecuteFile` (#25229 ↴)
- Make PowerShell class not affiliate with Runspace when declaring the `NoRunspaceAffinity` attribute (#18138 ↴)
- Add the `ValidateNotNullOrWhiteSpace` attribute (#17191 ↴) (Thanks @wmentha!)
- Add `sqlcmd` to the list for legacy argument passing (#18559 ↴)
- Add the function `cd~` (#18308 ↴) (Thanks @GigaScratch!)
- Fix array type parsing in generic types (#19205 ↴) (Thanks @MartinGC94!)
- Fix wildcard globbing in root of device paths (#19442 ↴) (Thanks @MartinGC94!)
- Add a public API for getting locations of `PSModulePath` elements (#19422 ↴)
- Fix incorrect string to type conversion (#19560 ↴) (Thanks @MartinGC94!)
- Fix slow execution when many breakpoints are used (#14953 ↴) (Thanks @nohwnd!)
- Remove code related to `#Requires -PSSnapin` (#19320 ↴)

Experimental Features

PowerShell 7.4 introduces the following experimental features:

- [PSFeedbackProvider](#) - Replaces the hard-coded suggestion framework with an extensible feedback provider.
 - This feature also adds the **FeedbackName**, **FeedbackText**, and **FeedbackAction** properties to `$PSStyle.Formatting` that allow you to change the formatting of feedback messages.
- [PSModuleAutoLoadSkipOfflineFiles](#) - Module discovery now skips over files that are marked by cloud providers as not fully on disk.
- [PSCmdWithArgs](#) - Add support for passing arguments to commands as a single string

The following experimental features became mainstream:

- [PSConstrainedAuditLogging](#)
- [PSCustomTableHeaderLabelDecoration](#)
- [PSNativeCommandErrorActionPreference](#)
- [PSNativeCommandPreserveBytePipe](#)
- [PSWindowsNativeCommandArgPassing](#)

PowerShell 7.4 changed the following experimental features:

- [PSCmdNotFoundSuggestion](#) - This feature now uses an extensible feedback provider rather than hard-coded suggestions ([#18726 ↗](#))

For more information about the Experimental Features, see [Using Experimental Features](#).

What's New in PowerShell 7.3

Article • 01/23/2025

PowerShell 7.3 is the next stable release, built on .NET 7.0.

PowerShell 7.3 includes the following features, updates, and breaking changes.

Breaking Changes and Improvements

- In this release, Windows APIs were updated or removed for compliance, which means that PowerShell 7.3 doesn't run on Windows 7. While Windows 7 is no longer supported, previous builds could run on Windows 7.
- PowerShell Direct for Hyper-V is only supported on Windows 10, version 1809 and higher.
- `Test-Connection` is broken due to an intentional [breaking change](#) in .NET 7. It's tracked by [#17018](#)
- Add `clean` block to script block as a peer to `begin`, `process`, and `end` to allow easy resource cleanup ([#15177](#))
- Change default for `$PSStyle.OutputRendering` to `Host`
- Make `Out-String` and `Out-File` keep string input unchanged ([#17455](#))
- Move the type data definition of `System.Security.AccessControl.ObjectSecurity` to the `Microsoft.PowerShell.Security` module ([#16355](#)) (Thanks @iSazonov!)
 - Before this change, a user doesn't need to explicitly import the `Microsoft.PowerShell.Security` module to use the code properties defined for an instance of `System.Security.AccessControl.ObjectSecurity`.
 - After this change, a user needs to explicitly import `Microsoft.PowerShell.Security` module in order to use those code properties and code methods.

Tab completion improvements

- PowerShell 7.3 includes PSReadLine 2.2.6, which enables Predictive IntelliSense by default. For more information, see [about_PSReadLine](#).
- Fix tab completion within the script block specified for the `ValidateScriptAttribute`. ([#14550](#)) (Thanks @MartinGC94!)
- Added tab completion for loop labels after `break`/`continue` ([#16438](#)) (Thanks @MartinGC94!)
- Improve Hashtable completion in multiple scenarios ([#16498](#)) (Thanks @MartinGC94!)

- Parameter splatting
- **Arguments** parameter for `Invoke-CimMethod`
- **FilterHashtable** parameter for `Get-WinEvent`
- **Property** parameter for the CIM cmdlets
- Removes duplicates from member completion scenarios
- Support forward slashes in network share (UNC path) completion (#17111 ↴) (Thanks @sba923!)
- Improve member autocompletion (#16504 ↴) (Thanks @MartinGC94!)
- Prioritize ValidateSet completions over Enums for parameters (#15257 ↴) (Thanks @MartinGC94!)
- Add type inference support for generic methods with type parameters (#16951 ↴) (Thanks @MartinGC94!)
- Improve type inference and completions (#16963 ↴) (Thanks @MartinGC94!)
 - Allows methods to be shown in completion results for `ForEach-Object -MemberName`
 - Prevents completion on expressions that return void like `([void]("")`)
 - Allows non-default Class constructors to show up when class completion is based on the AST
- Improve type inference for `$_` (#17716 ↴) (Thanks @MartinGC94!)
- Fix type inference for **ICollection** (#17752 ↴) (Thanks @MartinGC94!)
- Prevent braces from being removed when completing variables (#17751 ↴) (Thanks @MartinGC94!)
- Add completion for index expressions for dictionaries (#17619 ↴) (Thanks @MartinGC94!)
- Fix type completion for attribute tokens (#17484 ↴) (Thanks @MartinGC94!)
- Improve dynamic parameter tab completion (#17661 ↴) (Thanks @MartinGC94!)
- Avoid binding positional parameters when completing parameter in front of value (#17693 ↴) (Thanks @MartinGC94!)

Improved error handling

- Set `$?` correctly for command expression with redirections (#16046 ↴)
- Fix a casting error when using `$PSNativeCommandUseErrorActionPreference` (#15993 ↴)
- Make the native command error handling optionally honor `ErrorActionPreference` (#15897 ↴)
- Specify the executable path as `TargetObject` for non-zero exit code ErrorRecord (#16108 ↴) (Thanks @rkeithhill!)

Session and remoting improvements

- Add `-Options` to the PSRP over SSH commands to allow passing OpenSSH options directly ([#12802](#)) (Thanks @BrannenGH!)
- Add `-ConfigurationFile` parameter to `pwsh` to allow starting a new process with the session configuration defined in a `.pssc` file ([#17447](#))
- Add support for using `New-PSSessionConfigurationFile` on non-Windows platforms ([#17447](#))

Updated cmdlets

- Add `-HttpVersion` parameter to web cmdlets ([#15853](#)) (Thanks @hayhay27!)
- Add support to web cmdlets for open-ended input tags ([#16193](#)) (Thanks @farmerau!)
- Fix `ConvertTo-Json -Depth` to allow 100 at maximum ([#16197](#)) (Thanks @KevRitchie!)
- Improve variable handling when calling `Invoke-Command` with the `$Using:` expression ([#16113](#)) (Thanks @dwtaber!)
- Add `-StrictMode` to `Invoke-Command` to allow specifying strict mode when invoking command locally ([#16545](#)) (Thanks @Thomas-Yu!)
- Add `clean` block to script block as a peer to `begin`, `process`, and `end` to allow easy resource cleanup ([#15177](#))
- Add `-Amended` switch to `Get-CimClass` cmdlet ([#17477](#)) (Thanks @iSazonov)
- Changed `ConvertFrom-Json -AsHashtable` to use ordered hashtable ([#17405](#))
- Removed ANSI escape sequences in strings before sending to `Out-GridView` ([#17664](#))
- Added the `Milliseconds` parameter to `New-TimeSpan` ([#17621](#)) (Thanks @NoMoreFood!)
- Show optional parameters when displaying method definitions and overloads ([#13799](#)) (Thanks @eugenescmlv!)
- Allow commands to still be executed even if the current working directory no longer exists ([#17579](#))
- Add support for HTTPS with `Set-AuthenticodeSignature -TimeStampServer` ([#16134](#)) (Thanks @Ryan-Hutchison-USAF!)
- Render decimal numbers in a table using current culture ([#17650](#))
- Add type accelerator ordered for `OrderedDictionary` ([#17804](#)) (Thanks @fflaten!)
- Add `find.exe` to legacy argument binding behavior for Windows ([#17715](#))
- Add `-NoProfileLoadTime` switch to `pwsh` ([#17535](#)) (Thanks @rkeithhill!)

For a complete list of changes, see the [Change Log](#) in the GitHub repository.

Experimental Features

In PowerShell 7.3, following experimental features became mainstream:

- `PSAnsiRenderingFileInfo` - This feature adds the `$PSStyle.FileInfo` member and enables coloring of specific file types.
- `PSCleanBlock` - Adds `clean` block to script block as a peer to `begin`, `process`, and `end` to allow easy resource cleanup.
- `PSAMSIMethodInvocationLogging` - Extends the data sent to AMSI for inspection to include all invocations of .NET method members.
- `PSNativeCommandArgumentPassing` - PowerShell now uses the `ArgumentList` property of the `StartProcessInfo` object rather than the old mechanism of reconstructing a string when invoking a native executable.

PowerShell 7.3.1 adds `sqlcmd.exe` to the list of native commands in Windows that use the `Legacy` style of argument passing.

- `PSExec` - Adds the new `Switch-Process` cmdlet (alias `exec`) to provide `exec` compatibility for non-Windows systems.

PowerShell 7.3.1 changed the `exec` alias to a function that wraps `Switch-Process`. The function allows you to pass parameters to the native command that might have erroneously bound to the `WithCommand` parameter.

PowerShell 7.3 introduces the following experimental features:

- `PSNativeCommandErrorActionPreference` - Adds the `$PSNativeCommandUseErrorActionPreference` variable to enable errors produced by native commands to be PowerShell errors.

PowerShell 7.3 removed the following experimental features:

- `PSNativePSPPathResolution` experimental feature is no longer supported.
- `PSStrictModeAssignment` experimental feature is no longer supported.

For more information about the Experimental Features, see [Using Experimental Features](#).

What's New in PowerShell 7.2

Article • 01/23/2025

PowerShell 7.2 is the next Long Term Servicing (LTS) release is built on .NET 6.0.

PowerShell 7.2 includes the following features, updates, and breaking changes.

- New universal installer packages for most supported Linux distributions
- Microsoft Update support on Windows
- 2 new experimental features
 - Improved native command argument passing support
 - ANSI FileInfo color support
- Improved Tab Completions
- PSReadLine 2.1 with Predictive IntelliSense
- 7 experimental features promoted to mainstream and 1 removed
- Separating DSC from PowerShell 7 to enable future improvements
- Several breaking changes to improve usability

For a complete list of changes, see the [Change Log ↗](#) in the GitHub repository.

Installation updates

Check the installation instructions for your preferred operating system:

- [Windows](#)
- [macOS](#)
- [Linux](#)

Additionally, PowerShell 7.2 supports ARM64 versions of Windows and macOS and ARM32 and ARM64 versions of Debian and Ubuntu.

For up-to-date information about supported operating systems and support lifecycle, see the [PowerShell Support Lifecycle](#).

New universal install packages for Linux distributions

Previously, we created separate installer packages for each supported version of CentOS, RHEL, Debian, and Ubuntu. The universal installer package combines eight different packages into one, making installation on Linux simpler. The universal package installs the necessary dependencies for the target distribution and creates the platform-specific changes to make PowerShell work.

Microsoft Update support for Windows

PowerShell 7.2 add support for Microsoft Update. When you enable this feature, you'll get the latest PowerShell 7 updates in your traditional Windows Update (WU) management flow, whether that's with Windows Update for Business, WSUS, SCCM, or the interactive WU dialog in Settings.

The PowerShell 7.2 MSI package includes following command-line options:

- `USE_MU` - This property has two possible values:
 - `1` (default) - Opts into updating through Microsoft Update or WSUS
 - `0` - don't opt into updating through Microsoft Update or WSUS
- `ENABLE_MU`
 - `1` (default) - Opts into using Microsoft Update the Automatic Updates or Windows Update
 - `0` - don't opt into using Microsoft Update the Automatic Updates or Windows Update

Experimental Features

The following experimental features are now mainstream features in this release:

- `Microsoft.PowerShell.Utility.PSImportPSDataFileSkipLimitCheck` - see [Import-PowerShellDataFile](#)
- `Microsoft.PowerShell.Utility.PSManageBreakpointsInRunspace`
- `PSAnsiRendering` - see [about_ANSI_Terminals](#)
- `PSAnsiProgress` - see [about_ANSI_Terminals](#)
- `PSCultureInvariantReplaceOperator`
- `PSNotApplyErrorActionToStderr`
- `PSUnixFileStat`

The following experimental feature was added in this release:

- [PSNativeCommandArgumentPassing](#) - When this experimental feature is enabled PowerShell uses the `ArgumentList` property of the `StartProcessInfo` object rather than our current mechanism of reconstructing a string when invoking a native executable. This feature adds a new automatic variable `$PSNativeCommandArgumentPassing` that allows you to select the behavior at runtime.
- [PSAnsiRenderingFileInfo](#) - Allow ANSI color customization of file information.

- `PSLoadAssemblyFromNativeCode` - Exposes an API to allow assembly loading from native code.

For more information about the Experimental Features, see [Using Experimental Features](#).

Improved Tab Completions

PowerShell 7.2 includes several improvements to Tab Completion. These changes include bugfixes and improve usability.

- Fix tab completion for unlocalized about* topics (#15265) (Thanks @MartinGC94)
- Fix splatting being treated as positional parameter in completions (#14623) (Thanks @MartinGC94)
- Add completions for comment-based help keywords (#15337) (Thanks @MartinGC94)
- Add completion for Requires statements (#14596) (Thanks @MartinGC94)
- Added tab completion for View parameter of Format-* cmdlets (#14513) (Thanks @iSazonov)

PSReadLine 2.1 Predictive IntelliSense

PSReadLine 2.1 introduced `CommandPrediction` APIs that establish a framework for providing predictions for command-line completion. The API enables users to discover, edit, and execute full commands based on matching predictions from the user's history.

Predictive IntelliSense is disabled by default. To enable predictions, run the following command:

```
PowerShell  
Set-PSReadLineOption -PredictionSource History
```

Separating DSC from PowerShell 7 to enable future improvements

The PSDesiredStateConfiguration module was removed from the PowerShell 7.2 package and is now published to the PowerShell Gallery. This allows the PSDesiredStateConfiguration module to be developed independently of PowerShell and users can mix and match versions of PowerShell and PSDesiredStateConfiguration for

their environment. To install PSDesiredStateConfiguration 2.0.5 from the PowerShell Gallery:

```
PowerShell
```

```
Install-Module -Name PSDesiredStateConfiguration -Repository PSGallery -  
MaximumVersion 2.99
```

ⓘ Important

Be sure to include the parameter MaximumVersion or you could install version 3 (or higher) of PSDesireStateConfiguration that contains significant differences.

Engine updates

- Add `LoadAssemblyFromNativeMemory` function to load assemblies in memory from a native PowerShell host by awakecoding · Pull Request #14652

Breaking Changes and Improvements

- The PSDesiredStateConfiguration was removed from the PowerShell 7.2 package
- Make PowerShell Linux deb and RPM packages universal (#15109)
- Experimental feature `PSNativeCommandArgumentPassing`: Use ArgumentList for native executable invocation (#14692)
- Ensure `-PipelineVariable` is set for all output from script cmdlets (#12766)
- Emit warning if `ConvertToJson` exceeds `-Depth` value (#13692)
- Remove alias D of `-Directory` switch CL-General #15171
- Improve detection of mutable value types (#12495)
- Restrict `New-Object` in `NoLanguage` mode under lock down (#14140)
- Enforce AppLocker Deny configuration before Execution Policy Bypass configuration (#15035)
- Change `FileSystemInfo.Target` from a `CodeProperty` to an `AliasProperty` that points to `FileSystemInfo.LinkTarget` (#16165)

Migrating from Windows PowerShell 5.1 to PowerShell 7

Article • 04/02/2024

Designed for cloud, on-premises, and hybrid environments, PowerShell 7 is packed with enhancements and [new features](#).

- Installs and runs side-by-side with Windows PowerShell
- Improved compatibility with existing Windows PowerShell modules
- New language features, like ternary operators and `ForEach-Object -Parallel`
- Improved performance
- SSH-based remoting
- Cross-platform interoperability
- Support for Docker containers

PowerShell 7 works side-by-side with Windows PowerShell letting you easily test and compare between editions before deployment. Migration is simple, quick, and safe.

PowerShell 7 is supported on the following Windows operating systems:

- Windows 10, and 11
- Windows Server 2016, 2019, and 2022

PowerShell 7 also runs on macOS and several Linux distributions. For a list of supported operating systems and information about the support lifecycle, see the [PowerShell Support Lifecycle](#).

Installing PowerShell 7

For flexibility and to support the needs of IT, DevOps engineers, and developers, there are several options available to install PowerShell 7. In most cases, the installation options can be reduced to the following methods:

- Deploy PowerShell using the [MSI package](#)
- Deploy PowerShell using the [ZIP package](#)

Note

The MSI package can be deployed and updated with management products such as [Microsoft Configuration Manager](#). Download the packages from [GitHub Release page](#).

Deploying the MSI package requires Administrator permission. The ZIP package can be deployed by any user. The ZIP package is the easiest way to install PowerShell 7 for testing, before committing to a full installation.

You may also install PowerShell 7 via the Windows Store or [winget](#). For more information about both of these methods, see the detailed instructions in [Installing PowerShell on Windows](#).

Using PowerShell 7 side-by-side with Windows PowerShell 5.1

PowerShell 7 is designed to coexist with Windows PowerShell 5.1. The following features ensure that your investment in PowerShell is protected and your migration to PowerShell 7 is simple.

- Separate installation path and executable name
- Separate PSModulePath
- Separate profiles for each version
- Improved module compatibility
- New remoting endpoints
- Group policy support
- Separate Event logs

Differences in .NET versions

PowerShell 7.4 is built on .NET 8.0. Windows PowerShell 5.1 is built on .NET Framework 4.x. The differences between the .NET versions might affect the behavior of your scripts, especially if you are calling .NET method directly. For more information, [Differences between Windows PowerShell 5.1 and PowerShell 7.x](#).

Separate installation path and executable name

PowerShell 7 installs to a new directory, enabling side-by-side execution with Windows PowerShell 5.1.

Install locations by version:

- Windows PowerShell 5.1: `$Env:windir\System32\WindowsPowerShell\v1.0`
- PowerShell 6.x: `$Env:ProgramFiles\PowerShell\6`
- PowerShell 7: `$Env:ProgramFiles\PowerShell\7`

The new location is added to your PATH allowing you to run both Windows PowerShell 5.1 and PowerShell 7. If you're migrating from PowerShell 6.x to PowerShell 7, PowerShell 6 is removed and the PATH replaced.

In Windows PowerShell, the PowerShell executable is named `powershell.exe`. In version 6 and above, the executable is named `pwsh.exe`. The new name makes it easy to support side-by-side execution of both versions.

Separate PSModulePath

By default, Windows PowerShell and PowerShell 7 store modules in different locations.

PowerShell 7 combines those locations in the `$Env:PSModulePath` environment variable. When importing a module by name, PowerShell checks the location specified by `$Env:PSModulePath`. This allows PowerShell 7 to load both Core and Desktop modules.

 Expand table

Install Scope	Windows PowerShell 5.1	PowerShell 7.0
PowerShell modules	<code>\$Env:windir\system32\WindowsPowerShell\v1.0\Modules</code>	<code>\$Env:ProgramFiles\PowerShell\7\Modules</code>
User installed AllUsers scope	<code>\$Env:ProgramFiles\WindowsPowerShell\Modules</code>	<code>\$Env:ProgramFiles\PowerShell\Modules</code>
User installed CurrentUser scope	<code>\$HOME\Documents\WindowsPowerShell\Modules</code>	<code>\$HOME\Documents\PowerShell\Modules</code>

The following examples show the default values of `$Env:PSModulePath` for each version.

- For Windows PowerShell 5.1:

```
PowerShell  
$Env:PSModulePath -split ';'
```

```
Output  
C:\Users<user>\Documents\WindowsPowerShell\Modules  
C:\Program Files\WindowsPowerShell\Modules  
C:\WINDOWS\System32\WindowsPowerShell\v1.0\Modules
```

- For PowerShell 7:

```
PowerShell
```

```
$Env:PSModulePath -split ';'
```

Output

```
C:\Users\<user>\Documents\PowerShell\Modules
C:\Program Files\PowerShell\Modules
C:\Program Files\PowerShell\7\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\WINDOWS\System32\WindowsPowerShell\v1.0\Modules
```

Notice that PowerShell 7 includes the Windows PowerShell paths and the PowerShell 7 paths to provide autoloading of modules.

① Note

Additional paths may exist if you have changed the PSModulePath environment variable or installed custom modules or applications.

For more information, see [about_PSMODULEPATH](#).

For more information about Modules, see [about_Modules](#).

Separate profiles

A PowerShell profile is a script that executes when PowerShell starts. This script customizes your environment by adding commands, aliases, functions, variables, modules, and PowerShell drives. The profile script makes these customizations available in every session without having to manually recreate them.

The path to the location of the profile has changed in PowerShell 7.

- In Windows PowerShell 5.1, the location of the profile is `$HOME\Documents\WindowsPowerShell`.
- In PowerShell 7, the location of the profile is `$HOME\Documents\PowerShell`.

The profile filenames have also changed:

PowerShell

```
$PROFILE | Select-Object *Host* | Format-List
```

Output

```
AllUsersAllHosts      : C:\Program Files\PowerShell\7\profile.ps1
AllUsersCurrentHost   : C:\Program
Files\PowerShell\7\Microsoft.PowerShell_profile.ps1
```

```
CurrentUserAllHosts      : C:\Users\<user>\Documents\PowerShell\profile.ps1
CurrentUserCurrentHost : C:\Users\
<user>\Documents\PowerShell\Microsoft.PowerShell_profile.ps1
```

For more information [about_Profiles](#).

PowerShell 7 compatibility with Windows PowerShell 5.1 modules

Most of the modules you use in Windows PowerShell 5.1 already work with PowerShell 7, including Azure PowerShell and Active Directory. We're continuing to work with other teams to add native PowerShell 7 support for more modules including Microsoft Graph, Office 365, and others. For the current list of supported modules, see [PowerShell 7 module compatibility](#).

Note

On Windows, we've also added a **UseWindowsPowerShell** switch to `Import-Module` to ease the transition to PowerShell 7 for those using incompatible modules. For more information on this functionality, see [about_Windows_PowerShell_Compatibility](#).

PowerShell Remoting

PowerShell remoting lets you run any PowerShell command on one or more remote computers. You can establish persistent connections, start interactive sessions, and run scripts on remote computers.

WS-Management remoting

Windows PowerShell 5.1 and below use the WS-Management (WSMAN) protocol for connection negotiation and data transport. Windows Remote Management (WinRM) uses the WSMAN protocol. If WinRM has been enabled, PowerShell 7 uses the existing Windows PowerShell 5.1 endpoint named `Microsoft.PowerShell` for remoting connections. To update PowerShell 7 to include its own endpoint, run the `Enable-PSRemoting` cmdlet. For information about connecting to specific endpoints, see [WS-Management Remoting in PowerShell](#)

To use Windows PowerShell remoting, the remote computer must be configured for remote management. For more information, including instructions, see [About Remote Requirements](#).

For more information about working with remoting, see [About Remote](#)

SSH-based remoting

SSH-based remoting was added in PowerShell 6.x to support other operating systems that can't use Windows native components like WinRM. SSH remoting creates a PowerShell host process on the target computer as an SSH subsystem. For details and examples on setting up SSH-based remoting on Windows or Linux, see: [PowerShell remoting over SSH](#).

① Note

The PowerShell Gallery (PSGallery) contains a module and cmdlet that automatically configures SSH-based remoting. Install the `Microsoft.PowerShell.RemotingTools` module from the [PSGallery](#) and run the `Enable-SSH` cmdlet.

The `New-PSSession`, `Enter-PSSession`, and `Invoke-Command` cmdlets have new parameter sets to support SSH connections.

PowerShell

```
[ -HostName <string> ] [ -UserName <string> ] [ -KeyFilePath <string> ]
```

To create a remote session, specify the target computer with the **HostName** parameter and provide the user name with **UserName**. When running the cmdlets interactively, you're prompted for a password.

PowerShell

```
Enter-PSSession -HostName <Computer> -UserName <Username>
```

Alternatively, when using the **HostName** parameter, provide the username information followed by the at sign (@), followed by the computer name.

PowerShell

```
Enter-PSSession -HostName <Username>@<Computer>
```

You may set up SSH key authentication using a private key file with the **KeyFilePath** parameter. For more information, see [OpenSSH Key Management](#).

Group Policy supported

PowerShell includes Group Policy settings to help you define consistent option values for servers in an enterprise environment. These settings include:

- Console session configuration: Sets a configuration endpoint in which PowerShell is run.
- Turn on Module Logging: Sets the `LogPipelineExecutionDetails` property of modules.

- Turn on PowerShell Script Block Logging: Enables detailed logging of all PowerShell scripts.
- Turn on Script Execution: Sets the PowerShell execution policy.
- Turn on PowerShell Transcription: enables capturing of input and output of PowerShell commands into text-based transcripts.
- Set the default source path for Update-Help: Sets the source for Updatable Help to a directory, not the Internet.

For more information, see [about_Group_Policy_Settings](#).

PowerShell 7 includes Group Policy templates and an installation script in `$PSHOME`.

Group Policy tools use administrative template files (`.admx`, `.adml`) to populate policy settings in the user interface. This allows administrators to manage registry-based policy settings. The `InstallPSCorePolicyDefinitions.ps1` script installs PowerShell Administrative Templates on the local machine.

PowerShell

```
Get-ChildItem -Path $PSHOME -Filter *Core*Policy*
```

Output

Directory: C:\Program Files\PowerShell\7

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	2/27/2020 12:38 AM	15861	
	InstallPSCorePolicyDefinitions.ps1		
-a---	2/27/2020 12:28 AM	9675	
	PowerShellCoreExecutionPolicy.adml		
-a---	2/27/2020 12:28 AM	6201	
	PowerShellCoreExecutionPolicy.admx		

Separate Event Logs

Windows PowerShell and PowerShell 7 log events to separate event logs. Use the following command to get a list of the PowerShell logs.

PowerShell

```
Get-WinEvent -ListLog *PowerShell*
```

For more information, see [about_Logging_Windows](#).

Improved editing experience with Visual Studio Code

Visual Studio Code (VSCode)  with the [PowerShell Extension](#)  is the supported scripting environment for PowerShell 7. The Windows PowerShell Integrated Scripting Environment (ISE) only supports Windows PowerShell.

The updated PowerShell extension includes:

- New ISE compatibility mode
- PSReadLine in the Integrated Console, including syntax highlighting, multi-line editing, and back search
- Stability and performance improvements
- New CodeLens integration
- Improved path autocomplete

To make the transition to Visual Studio Code easier, use the **Enable ISE Mode** function available in the **Command Palette**. This function switches VSCode into an ISE-style layout. The ISE-style layout gives you all the new features and capabilities of PowerShell in a familiar user experience.

To switch to the new ISE layout, press `Ctrl + Shift + P` to open the **Command Palette**, type `PowerShell` and select **PowerShell: Enable ISE Mode**.

To set the layout to the original layout, open the **Command Palette**, select **PowerShell: Disable ISE Mode (restore to defaults)**.

For details about customizing the VSCode layout to ISE, see [How to Replicate the ISE Experience in Visual Studio Code](#)

Note

There are no plans to update the ISE with new features. In the latest versions of Windows 10 or Windows Server 2019 and higher, the ISE is now a user-uninstallable feature. There are no plans to permanently remove the ISE. The PowerShell Team and its partners are focused on improving the scripting experience in the PowerShell extension for Visual Studio Code.

Next Steps

Armed with the knowledge to effectively migrate, [install PowerShell 7](#) now!

Differences between Windows PowerShell 5.1 and PowerShell 7.x

Article • 04/02/2024

Windows PowerShell 5.1 is built on top of the .NET Framework v4.5. With the release of PowerShell 6.0, PowerShell became an open source project built on .NET Core 2.0. Moving from the .NET Framework to .NET Core allowed PowerShell to become a cross-platform solution. PowerShell runs on Windows, macOS, and Linux.

There are few differences in the PowerShell language between Windows PowerShell and PowerShell. The most notable differences are in the availability and behavior of PowerShell cmdlets between Windows and non-Windows platforms and the changes that stem from the differences between the .NET Framework and .NET Core.

This article summarizes the significant differences and breaking changes between Windows PowerShell and the current version of PowerShell. This summary does not include new features or cmdlets that have been added. Nor does this article discuss what changed between versions. The goal of this article is to present the current state of PowerShell and how that is different from Windows PowerShell. For a detailed discussion of changes between versions and the addition of new features, see the [What's New](#) articles for each version.

- [What's new in PowerShell 7.5](#)
- [What's new in PowerShell 7.4](#)
- [What's new in PowerShell 7.3](#)
- [What's new in PowerShell 7.2](#)
- [What's new in PowerShell 7.1](#)
- [What's new in PowerShell 7.0](#)
- [What's new in PowerShell 6.x](#)

.NET Framework vs .NET Core

PowerShell on Linux and macOS uses .NET core, which is a subset of the full .NET Framework on Microsoft Windows. This is significant because PowerShell provides direct access to the underlying framework types and methods. As a result, scripts that run on Windows may not run on non-Windows platforms because of the differences in the frameworks. For more information about changes in .NET Core, see [Breaking changes for migration from .NET Framework to .NET Core](#).

Each new release of PowerShell is built on a newer version of .NET. There can be breaking changes in .NET that affect PowerShell.

- PowerShell 7.5 - Built on .NET 9.0
- PowerShell 7.4 - Built on .NET 8.0
- PowerShell 7.3 - Built on .NET 7.0
- PowerShell 7.2 (LTS-current) - Built on .NET 6.0 (LTS-current)
- PowerShell 7.1 - Built on .NET 5.0
- PowerShell 7.0 (LTS) - Built on .NET Core 3.1 (LTS)
- PowerShell 6.2 - Built on .NET Core 2.1
- PowerShell 6.1 - Built on .NET Core 2.1
- PowerShell 6.0 - Built on .NET Core 2.0

With the advent of [.NET Standard 2.0](#), PowerShell can load many traditional Windows PowerShell modules without modification. Additionally, PowerShell 7 includes a Windows PowerShell Compatibility feature that allows you to use Windows PowerShell modules that still require the full framework.

For more information see:

- [about_Windows_PowerShell_Compatibility](#)
- [PowerShell 7 module compatibility](#)

Be aware of .NET method changes

While .NET method changes are not specific to PowerShell, they can affect your scripts, especially if you are calling .NET methods directly. Also, there might be new overloads for constructors. This can have an impact on how you create objects using `New-Object` or the `[type]::new()` method.

For example, .NET added overloads to the `[System.String]::Split()` method that aren't available in .NET Framework 4.5. The following list shows the overloads for the `Split()` method available in Windows PowerShell 5.1:

```
PowerShell
PS> "" .Split
OverloadDefinitions
-----
string[] Split(Params char[] separator)
string[] Split(char[] separator, int count)
string[] Split(char[] separator, System.StringSplitOptions options)
string[] Split(char[] separator, int count, System.StringSplitOptions options)
```

```
string[] Split(string[] separator, System.StringSplitOptions options)
string[] Split(string[] separator, int count, System.StringSplitOptions
options)
```

The following list shows the overloads for the `Split()` method available in PowerShell 7:

```
PowerShell

"".Split

OverloadDefinitions
-----
string[] Split(char separator, System.StringSplitOptions options)
string[] Split(char separator, int count, System.StringSplitOptions options)
string[] Split(Params char[] separator)
string[] Split(char[] separator, int count)
string[] Split(char[] separator, System.StringSplitOptions options)
string[] Split(char[] separator, int count, System.StringSplitOptions
options)
string[] Split(string separator, System.StringSplitOptions options)
string[] Split(string separator, int count, System.StringSplitOptions
options)
string[] Split(string[] separator, System.StringSplitOptions options)
string[] Split(string[] separator, int count, System.StringSplitOptions
options)
```

In Windows PowerShell 5.1, you could pass a character array (`char[]`) to the `Split()` method as a `string`. The method splits the target string at any occurrence of a character in the array. The following command splits the target string in Windows PowerShell 5.1, but not in PowerShell 7:

```
PowerShell

# PowerShell 7 example
"1111p2222q3333".Split('pq')
```

Output

```
1111p2222q3333
```

To bind to the correct overload, you must typecast the string to a character array:

```
PowerShell

# PowerShell 7 example
"1111p2222q3333".Split([char[]]'pq')
```

Output

```
1111  
2222  
3333
```

Modules no longer shipped with PowerShell

For various compatibility reasons, the following modules are no longer included in PowerShell.

- ISE
- Microsoft.PowerShell.LocalAccounts
- Microsoft.PowerShell.ODataUtils
- Microsoft.PowerShell.Operation.Validation
- PSScheduledJob
- PSWorkflow
- PSWorkflowUtility

PowerShell Workflow

[PowerShell Workflow](#) is a feature in Windows PowerShell that builds on top of [Windows Workflow Foundation \(WF\)](#) that enables the creation of robust runbooks for long-running or parallelized tasks.

Due to the lack of support for Windows Workflow Foundation in .NET Core, we removed PowerShell Workflow from PowerShell.

In the future, we would like to enable native parallelism/concurrency in the PowerShell language without the need for PowerShell Workflow.

If there is a need to use checkpoints to resume a script after the OS restarts, we recommend using Task Scheduler to run a script on OS startup, but the script would need to maintain its own state (like persisting it to a file).

Cmdlets removed from PowerShell

For the modules that are included in PowerShell, the following cmdlets were removed from PowerShell for various compatibility reasons or the use of unsupported APIs.

CimCmdlets

- `Export-BinaryMiLog`

Microsoft.PowerShell.Core

- `Add-PSSnapin`
- `Export-Console`
- `Get-PSSnapin`
- `Remove-PSSnapin`
- `Resume-Job`
- `Suspend-Job`

Microsoft.PowerShell.Diagnostics

- `Export-Counter`
- `Import-Counter`

Microsoft.PowerShell.Management

- `Add-Computer`
- `Checkpoint-Computer`
- `Clear-EventLog`
- `Complete-Transaction`
- `Disable-ComputerRestore`
- `Enable-ComputerRestore`
- `Get-ComputerRestorePoint`
- `Get-ControlPanelItem`
- `Get-EventLog`
- `Get-Transaction`
- `Get-WmiObject`
- `Invoke-WmiMethod`
- `Limit-EventLog`
- `New-EventLog`
- `New-WebServiceProxy`
- `Register-WmiEvent`
- `Remove-Computer`
- `Remove-EventLog`
- `Remove-WmiObject`
- `Reset-ComputerMachinePassword`
- `Restore-Computer`
- `Set-WmiInstance`

- `Show-ControlPanelItem`
- `Show-EventLog`
- `Start-Transaction`
- `Test-ComputerSecureChannel`
- `Undo-Transaction`
- `Use-Transaction`
- `Write-EventLog`

Microsoft.PowerShell.Utility

- `Convert-String`
- `ConvertFrom-String`

PSDesiredStateConfiguration

- `Disable-DscDebug`
- `Enable-DscDebug`
- `Get-DscConfiguration`
- `Get-DscConfigurationStatus`
- `Get-DscLocalConfigurationManager`
- `Publish-DscConfiguration`
- `Remove-DscConfigurationDocument`
- `Restore-DscConfiguration`
- `Set-DscLocalConfigurationManager`
- `Start-DscConfiguration`
- `Stop-DscConfiguration`
- `Test-DscConfiguration`
- `Update-DscConfiguration`

WMI v1 cmdlets

The following WMI v1 cmdlets were removed from PowerShell:

- `Register-WmiEvent`
- `Set-WmiInstance`
- `Invoke-WmiMethod`
- `Get-WmiObject`
- `Remove-WmiObject`

The CimCmdlets module (aka WMI v2) cmdlets perform the same function and provide new functionality and a redesigned syntax.

New-WebServiceProxy cmdlet removed

.NET Core does not support the Windows Communication Framework, which provide services for using the SOAP protocol. This cmdlet was removed because it requires SOAP.

*-Transaction cmdlets removed

These cmdlets had very limited usage. The decision was made to discontinue support for them.

- `Complete-Transaction`
- `Get-Transaction`
- `Start-Transaction`
- `Undo-Transaction`
- `Use-Transaction`

*-EventLog cmdlets

Due to the use of unsupported APIs, the `*-EventLog` cmdlets have been removed from PowerShell. `Get-WinEvent` and `New-WinEvent` are available to get and create events on Windows.

Cmdlets that use the Windows Presentation Framework (WPF)

.NET Core 3.1 added support for WPF, so the release of PowerShell 7.0 restored the following Windows-specific features:

- The `Show-Command` cmdlet
- The `Out-GridView` cmdlet
- The `ShowWindow` parameter of `Get-Help`

PowerShell Desired State Configuration (DSC) changes

`Invoke-DscResource` was restored as an experimental feature in PowerShell 7.0.

Beginning with PowerShell 7.2, the PSDesiredStateConfiguration module has been removed from PowerShell and has been published to the PowerShell Gallery. For more information, see the [announcement](#) in the PowerShell Team blog.

PowerShell executable changes

Renamed `powershell.exe` to `pwsh.exe`

The binary name for PowerShell has been changed from `powershell(.exe)` to `pwsh(.exe)`. This change provides a deterministic way for users to run PowerShell on machines and support side-by-side installations of Windows PowerShell and PowerShell.

Additional changes to `pwsh(.exe)` from `powershell.exe`:

- Changed the first positional parameter from `-Command` to `-File`. This change fixes the usage of `#!` (aka as a shebang) in PowerShell scripts that are being executed from non-PowerShell shells on non-Windows platforms. It also means that you can run commands like `pwsh foo.ps1` or `pwsh fooScript` without specifying `-File`. However, this change requires that you explicitly specify `-c` or `-Command` when trying to run commands like `pwsh.exe -Command Get-Command`.
- `pwsh` accepts the `-i` (or `-Interactive`) switch to indicate an interactive shell. This allows PowerShell to be used as a default shell on Unix platforms.
- Removed parameters `-ImportSystemModules` and `-PSConsoleFile` from `pwsh.exe`.
- Changed `pwsh -Version` and built-in help for `pwsh.exe` to align with other native tools.
- Invalid argument error messages for `-File` and `-Command` and exit codes consistent with Unix standards
- Added `-WindowStyle` parameter on Windows. Similarly, package-based installations updates on non-Windows platforms are in-place updates.

The shortened name is also consistent with naming of shells on non-Windows platforms.

Support running a PowerShell script with bool parameter

Previously, using `pwsh.exe` to execute a PowerShell script using `-File` provided no way to pass `$true/$false` as parameter values. Support for `$true/$false` as parsed values to parameters was added. Switch values are also supported.

Improved backwards compatibility with Windows PowerShell

For Windows, a new switch parameter `UseWindowsPowerShell` is added to `Import-Module`. This switch creates a proxy module in PowerShell 7 that uses a local Windows PowerShell process to implicitly run any cmdlets contained in that module. For more information, see [Import-Module](#).

For more information on which Microsoft modules work with PowerShell 7.0, see the [Module Compatibility Table](#).

Microsoft Update support for Windows

PowerShell 7.2 added support for Microsoft Update. When you enable this feature, you'll get the latest PowerShell 7 updates in your traditional Windows Update (WU) management flow, whether that's with Windows Update for Business, WSUS, SCCM, or the interactive WU dialog in Settings.

The PowerShell 7.2 MSI package includes following command-line options:

- `USE_MU` - This property has two possible values:
 - `1` (default) - Opts into updating through Microsoft Update or WSUS
 - `0` - Do not opt into updating through Microsoft Update or WSUS
- `ENABLE_MU`
 - `1` (default) - Opts into using Microsoft Update the Automatic Updates or Windows Update
 - `0` - Do not opt into using Microsoft Update the Automatic Updates or Windows Update

Engine changes

Support PowerShell as a default Unix shell

On Unix, it is a convention for shells to accept `-i` for an interactive shell and many tools expect this behavior (`script` for example, and when setting PowerShell as the default shell) and calls the shell with the `-i` switch. This change is breaking in that `-i` previously could be used as short hand to match `-InputFormat`, which now needs to be `-in`.

Custom snap-ins

[PowerShell snap-ins](#) are a predecessor to PowerShell modules that do not have widespread adoption in the PowerShell community.

Due to the complexity of supporting snap-ins and their lack of usage in the community, we no longer support custom snap-ins in PowerShell.

Experimental feature flags

PowerShell 6.2 enabled support for [Experimental Features](#). This allows PowerShell developers to deliver new features and get feedback before the design is complete. This way we avoid making breaking changes as the design evolves.

Use `Get-ExperimentalFeature` to get a list of available experimental features. You can enable or disable these features with `Enable-ExperimentalFeature` and `Disable-ExperimentalFeature`.

Load assembly from module base path before trying to load from the GAC

Previously, when a binary module has the module assembly in GAC, we loaded the assembly from GAC before trying to load it from module base path.

Skip null-element check for collections with a value-type element type

For the `Mandatory` parameter and `ValidateNotNull` and `ValidateNotNullOrEmpty` attributes, skip the null-element check if the collection's element type is value type.

Preserve `$?` for ParenExpression, SubExpression and ArrayExpression

This PR alters the way we compile subpipelines `(...)`, subexpressions `$(...)` and array expressions `@()` so that `$?` is not automatically `true`. Instead the value of `$?` depends on the result of the pipeline or statements executed.

Fix `$?` to not be `$false` when native command writes to `stderr`

`$?` is not set to `$false` when native command writes to `stderr`. It is common for native commands to write to `stderr` without intending to indicate a failure. `$?` is set to `$false` only when the native command has a non-zero exit code.

Make `$ErrorActionPreference` not affect `stderr` output of native commands

It is common for native commands to write to `stderr` without intending to indicate a failure. With this change, `stderr` output is still captured in `ErrorRecord` objects, but the runtime no longer applies `$ErrorActionPreference` if the `ErrorRecord` comes from a native command.

Change `$OutputEncoding` to use `UTF-8 NoBOM` encoding rather than ASCII

The previous encoding, ASCII (7-bit), would result in incorrect alteration of the output in some cases. Making `UTF-8 NoBOM` the default preserves Unicode output with an encoding supported by most tools and operating systems.

Unify cmdlets with parameter `-Encoding` to be of type `System.Text.Encoding`

The `-Encoding` value `Byte` has been removed from the `FileSystem` provider cmdlets. A new parameter, `-AsByteStream`, is now used to specify that a byte stream is required as input or that the output is a stream of bytes.

Change `New-ModuleManifest` encoding to `UTF8NoBOM` on non-Windows platforms

Previously, `New-ModuleManifest` creates `psd1` manifests in UTF-16 with BOM, creating a problem for Linux tools. This breaking change changes the encoding of `New-ModuleManifest` to be UTF (no BOM) in non-Windows platforms.

Remove `AllScope` from most default aliases

To speed up scope creation, `AllScope` was removed from most default aliases. `AllScope` was left for a few frequently used aliases where the lookup was faster.

`-Verbose` and `-Debug` no longer overrides `$ErrorActionPreference`

Previously, if `-Verbose` or `-Debug` were specified, it overrode the behavior of `$ErrorActionPreference`. With this change, `-Verbose` and `-Debug` no longer affect the behavior of `$ErrorActionPreference`.

Also, the `-Debug` parameter sets `$DebugPreference` to **Continue** instead of **Inquire**.

Make `$PSCulture` consistently reflect in-session culture changes

In Windows PowerShell, the current culture value is cached, which can allow the value to get out of sync with the culture is change after session-startup. This caching behavior is fixed in PowerShell core.

Allow explicitly specified named parameter to supersede the same one from hashtable splatting

With this change, the named parameters from splatting are moved to the end of the parameter list so that they are bound after all explicitly specified named parameters are bound. Parameter binding for simple functions doesn't throw an error when a specified named parameter cannot be found. Unknown named parameters are bound to the `$args` parameter of the simple function. Moving splatting to the end of the argument list changes the order the parameters appears in `$args`.

For example:

```
PowerShell

function SimpleTest {
    param(
        $Name,
        $Path
    )
    "Name: $Name; Path: $Path; Args: $args"
}
```

In the previous behavior, **MyPath** is not bound to `-Path` because it's the third argument in the argument list. ## So it ends up being stuffed into '\$args' along with `Blah = "World"`

```
PowerShell

PS> $hash = @{ Name = "Hello"; Blah = "World" }
PS> SimpleTest @hash "MyPath"
```

```
Name: Hello; Path: ; Args: -Blah: World MyPath
```

With this change, the arguments from `@hash` are moved to the end of the argument list. **MyPath** becomes the first argument in the list, so it is bound to `-Path`.

PowerShell

```
PS> SimpleTest @hash "MyPath"
Name: Hello; Path: MyPath; Args: -Blah: World
```

Language changes

Null-coalescing operator `??`

The null-coalescing operator `??` returns the value of its left-hand operand if it isn't null. Otherwise, it evaluates the right-hand operand and returns its result. The `??` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

PowerShell

```
$x = $null
$x ?? 100
```

Output

```
100
```

In the following example, the right-hand operand won't be evaluated.

PowerShell

```
[string] $todaysDate = '1/10/2020'
$todaysDate ?? (Get-Date).ToString()
```

Output

```
1/10/2020
```

Null-coalescing assignment operator `??=`

The null-coalescing assignment operator `??=` assigns the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to null. The `??` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

PowerShell

```
$x = $null  
$x ??= 100  
$x
```

Output

```
100
```

In the following example, the right-hand operand won't be evaluated.

PowerShell

```
[string] $todaysDate = '1/10/2020'  
$todaysDate ??= (Get-Date).ToShortDateString()
```

Output

```
1/10/2020
```

Null-conditional operators

ⓘ Note

This feature was moved from experimental to mainstream in PowerShell 7.1.

A null-conditional operator applies a member access, `?.`, or element access, `?[]`, operation to its operand only if that operand evaluates to non-null; otherwise, it returns null.

Since PowerShell allows `?` to be part of the variable name, formal specification of the variable name is required for using these operators. So it is required to use `{}` around the variable names like `${a}` or when `?` is part of the variable name `${a?}`.

In the following example, the value of `PropName` is returned.

PowerShell

```
$a = @{ PropName = 100 }
${a}?.PropName
```

Output

```
100
```

The following example will return null, without trying to access the member name **PropName**.

PowerShell

```
$a = $null
${a}?.PropName
```

Similarly, the value of the element will be returned.

PowerShell

```
$a = 1..10
${a}?[0]
```

Output

```
1
```

And when the operand is null, the element isn't accessed and null is returned.

PowerShell

```
$a = $null
${a}?[0]
```

ⓘ Note

The variable name syntax of `${<name>}` should not be confused with the `$()` subexpression operator. For more information, see Variable name section of [about_Variables](#).

Added & operator for job control

Putting & at the end of a pipeline causes the pipeline to be run as a PowerShell job. When a pipeline is backgrounded, a job object is returned. Once the pipeline is running as a job, all of the standard `*-Job` cmdlets can be used to manage the job. Variables (ignoring process-specific variables) used in the pipeline are automatically copied to the job so `Copy-Item $foo $bar &` just works. The job is also run in the current directory instead of the user's home directory.

New methods/properties on `PSCustomObject`

We've added new methods and properties to `PSCustomObject`. `PSCustomObject` now includes a `Count / Length` property like other objects.

PowerShell

```
$PSCustomObject = [pscUSTOMOBJECT]@{foo = 1}  
$PSCustomObject.Length
```

Output

```
1
```

PowerShell

```
$PSCustomObject.Count
```

Output

```
1
```

This work also includes `ForEach` and `Where` methods that allow you to operate and filter on `PSCustomObject` items:

PowerShell

```
$PSCustomObject.ForEach({$_.foo + 1})
```

Output

PowerShell

```
$PSCustomObject.Where({$_.foo -gt 0})
```

Output

```
foo
---
1
```

Conversions from PSMethod to Delegate

You can convert a `PSMethod` into a delegate. This allows you to do things like passing `PSMethod [M]::DoubleStrLen` as a delegate value into `[M]::AggregateString`:

PowerShell

```
class M {
    static [int] DoubleStrLen([string] $value) { return 2 * $value.Length }

    static [long] AggregateString([string[]] $values, [Func[string, int]] $selector) {
        [long] $res = 0
        foreach($s in $values){
            $res += $selector.Invoke($s)
        }
        return $res
    }
}

[M]::AggregateString((gci).Name, [M]::DoubleStrLen)
```

String comparison behavior changed in PowerShell 7.1

PowerShell 7.1 is built on .NET 5.0, which introduced the following breaking change:

- [Behavior changes when comparing strings on .NET 5+](#)

As of .NET 5.0, culture invariant string comparisons ignore non-printing control characters.

For example, the following two strings are considered to be identical:

PowerShell

```
# Escape sequence ``a`` is Ctrl-G or [char]7
'Food' -eq "Foo`ad"
```

Output

```
True
```

New cmdlets

New Get-Uptime cmdlet

The [Get-Uptime](#) cmdlet returns the time elapsed since the last boot of the operating system. The cmdlet was introduced in PowerShell 6.0.

New Remove-Alias cmdlet

The [Remove-Alias](#) cmdlet removes an alias from the current PowerShell session. The cmdlet was introduced in PowerShell 6.0.

New cmdlet Remove-Service

The [Remove-Service](#) cmdlet removes a Windows service in the registry and in the service database. The [Remove-Service](#) cmdlet was introduced in PowerShell 6.0.

New Markdown cmdlets

Markdown is a standard for creating readable plaintext documents with basic formatting that can be rendered into HTML.

The following cmdlets were added in PowerShell 6.1:

- [ConvertFrom-Markdown](#) - Convert the contents of a string or a file to a `MarkdownInfo` object.
- [Get-MarkdownOption](#) - Returns the current colors and styles used for rendering Markdown content in the console.
- [Set-MarkdownOption](#) - Sets the colors and styles used for rendering Markdown content in the console.
- [Show-Markdown](#) - Displays Markdown content in the console or as HTML

New Test-Json cmdlet

The [Test-Json](#) cmdlet tests whether a string is a valid JavaScript Object Notation (JSON) document and can optionally verify that JSON document against a provided schema.

This cmdlet was introduced in PowerShell 6.1

New cmdlets to support Experimental Features

The following cmdlets were added in PowerShell 6.2 to support Experimental Features.

- [Disable-ExperimentalFeature](#)
- [Enable-ExperimentalFeature](#)
- [Get-ExperimentalFeature](#)

New Join-String cmdlet

The [Join-String](#) cmdlet combines objects from the pipeline into a single string. This cmdlet was added in PowerShell 6.2.

New view ConciseView and cmdlet Get-Error

PowerShell 7.0 enhances the display of error messages to improve the readability of interactive and script errors with a new default view, **ConciseView**. The views are user-selectable through the preference variable `$ErrorView`.

With **ConciseView**, if an error is not from a script or parser error, then it's a single line error message:

```
PowerShell
Get-ChildItem -Path C:\NotReal
Output
Get-ChildItem: Cannot find path 'C:\NotReal' because it does not exist
```

If the error occurs during script execution or is a parsing error, PowerShell returns a multiline error message that contains the error, a pointer, and an error message showing where the error is in that line. If the terminal doesn't support ANSI color escape sequences (VT100), then colors are not displayed.

The default view in PowerShell 7 is **ConciseView**. The previous default view was **NormalView** and you can select this by setting the preference variable `$ErrorView`.

```
PowerShell
```

```
$ErrorView = 'NormalView' # Sets the error view to NormalView  
$ErrorView = 'ConciseView' # Sets the error view to ConciseView
```

ⓘ Note

A new property **ErrorAccentColor** is added to `$Host.PrivateData` to support changing the accent color of the error message.

The new `Get-Error` cmdlet provides a complete detailed view of the fully qualified error when desired. By default the cmdlet displays the full details, including inner exceptions, of the last error that occurred.

The `Get-Error` cmdlet supports input from the pipeline using the built-in variable `$Error`. `Get-Error` displays all piped errors.

```
PowerShell
```

```
$Error | Get-Error
```

The `Get-Error` cmdlet supports the **Newest** parameter, allowing you to specify how many errors from the current session you wish displayed.

```
PowerShell
```

```
Get-Error -Newest 3 # Displays the 1st three errors that occurred in the session
```

For more information, see [Get-Error](#).

Cmdlet changes

Parallel execution added to `ForEach-Object`

Beginning in PowerShell 7.0, the `ForEach-Object` cmdlet, which iterates items in a collection, now has built-in parallelism with the new **Parallel** parameter.

By default, parallel script blocks use the current working directory of the caller that started the parallel tasks.

This example retrieves 50,000 log entries from 5 system logs on a local Windows machine:

```
PowerShell

$logNames = 'Security','Application','System','Windows
PowerShell','Microsoft-Windows-Store/Operational'

$logEntries = $logNames | ForEach-Object -Parallel {
    Get-WinEvent -LogName $_ -MaxEvents 10000
} -ThrottleLimit 5

$logEntries.Count

50000
```

The **Parallel** parameter specifies the script block that is run in parallel for each input log name.

The new **ThrottleLimit** parameter limits the number of script blocks running in parallel at a given time. The default is 5.

Use the `$_` variable to represent the current input object in the script block. Use the `Using:` scope modifier to pass variable references to the running script block.

For more information, see [ForEach-Object](#).

Check `system32` for compatible built-in modules on Windows

In the Windows 10 1809 update and Windows Server 2019, we updated a number of built-in PowerShell modules to mark them as compatible with PowerShell.

When PowerShell starts up, it automatically includes `$windir\System32` as part of the `PSModulePath` environment variable. However, it only exposes modules to `Get-Module` and `Import-Module` if its `CompatiblePSEdition` is marked as compatible with `Core`.

You can override this behavior to show all modules using the `-SkipEditionCheck` switch parameter. We've also added a `PSEdition` property to the table output.

-1p alias for all `-LiteralPath` parameters

We created a standard parameter alias `-1p` for all the built-in PowerShell cmdlets that have a `-LiteralPath` parameter.

Fix `Get-Item -LiteralPath a*b` if `a*b` doesn't actually exist to return error

Previously, `-LiteralPath` given a wildcard would treat it the same as `-Path` and if the wildcard found no files, it would silently exit. Correct behavior should be that `-LiteralPath` is literal so if the file doesn't exist, it should error. Change is to treat wildcards used with `-Literal` as literal.

Set working directory to current directory in `Start-Job`

The `Start-Job` cmdlet now uses the current directory as the working directory for the new job.

Remove `-Protocol` from `*-Computer` cmdlets

Due to issues with RPC remoting in CoreFX (particularly on non-Windows platforms) and ensuring a consistent remoting experience in PowerShell, the `-Protocol` parameter was removed from the `*-Computer` cmdlets. DCOM is no longer supported for remoting.

The following cmdlets only support WSMAN remoting:

- `Rename-Computer`
- `Restart-Computer`
- `Stop-Computer`

Remove `-ComputerName` from `*-Service` cmdlets

In order to encourage the consistent use of PSRP, the `-ComputerName` parameter was removed from `*-Service` cmdlets.

Fix `Get-Content -Delimiter` to not include the delimiter in the returned lines

Previously, the output while using `Get-Content -Delimiter` was inconsistent and inconvenient as it required further processing of the data to remove the delimiter. This change removes the delimiter in returned lines.

Changes to `Format-Hex`

The `-Raw` parameter is now a "no-op" (in that it does nothing). Going forward all output is displayed with a true representation of numbers that includes all of the bytes for its type. This is what the `-Raw` parameter was doing prior to this change.

Typo fix in `Get-ComputerInfo` property name

`BiosSerialNumber` was misspelled as `BiosSeralNumber` and has been changed to the correct spelling.

Add `Get-StringHash` and `Get-FileHash` cmdlets

This change is that some hash algorithms are not supported by CoreFX, therefore they are no longer available:

- `MACTripleDES`
- `RIPEMD160`

Add validation on `Get-*` cmdlets where passing `$null` returns all objects instead of error

Passing `$null` to any of the following now throws an error:

- `Get-Credential -UserName`
- `Get-Event -SourceIdentifier`
- `Get-EventSubscriber -SourceIdentifier`
- `Get-Help -Name`
- `Get-PSBreakpoint -Script`
- `Get-PSPowerShellProvider -PSProvider`
- `Get-PSSessionConfiguration -Name`
- `Get-Runspace -Name`
- `Get-RunspaceDebug -RunspaceName`
- `Get-Service -Name`
- `Get-TraceSource -Name`
- `Get-Variable -Name`

Add support for the W3C Extended Log File Format in `Import-Csv`

Previously, the `Import-Csv` cmdlet cannot be used to directly import the log files in W3C extended log format and additional action would be required. With this change, W3C extended log format is supported.

`Import-Csv` applies `pstypenames` upon import when type information is present in the CSV

Previously, objects exported using `Export-Csv` with `TypeInformation` imported with `ConvertFrom-Csv` were not retaining the type information. This change adds the type information to `pstypenames` member if available from the CSV file.

-NoTypeInformation is the default on `Export-Csv`

Previously, the `Export-Csv` cmdlet would output a comment as the first line containing the type name of the object. The change excludes the type information by default because it's not understood by most CSV tools. This change was made to address customer feedback.

Use `-IncludeTypeInformation` to retain the previous behavior.

Allow `*` to be used in registry path for `Remove-Item`

Previously, `-LiteralPath` given a wildcard would treat it the same as `-Path` and if the wildcard found no files, it would silently exit. Correct behavior should be that `-LiteralPath` is literal so if the file doesn't exist, it should error. Change is to treat wildcards used with `-Literal` as literal.

Group-Object now sorts the groups

As part of the performance improvement, `Group-Object` now returns a sorted listing of the groups. Although you should not rely on the order, you could be broken by this change if you wanted the first group. We decided that this performance improvement was worth the change since the impact of being dependent on previous behavior is low.

Standard deviation in `Measure-Object`

The output from `Measure-Object` now includes a `StandardDeviation` property.

```
Get-Process | Measure-Object -Property CPU -AllStats
```

Output

```
Count          : 308
Average       : 31.3720576298701
Sum           : 9662.59375
Maximum       : 4416.046875
Minimum       :
StandardDeviation : 264.389544720926
Property      : CPU
```

Get-PfxCertificate -Password

`Get-PfxCertificate` now has the `Password` parameter, which takes a `SecureString`. This allows you to use it non-interactively:

PowerShell

```
$certFile = '\\server\share\pwd-protected.pfx'
$certPass = Read-Host -AsSecureString -Prompt 'Enter the password for
certificate: '

$certThumbPrint = (Get-PfxCertificate -FilePath $certFile -Password
$certPass ).ThumbPrint
```

Removal of the `more` function

In the past, PowerShell shipped a function on Windows called `more` that wrapped `more.com`. That function has now been removed.

Also, the `help` function changed to use `more.com` on Windows, or the system's default pager specified by `$Env:PAGER` on non-Windows platforms.

`cd DriveName:` now returns users to the current working directory in that drive

Previously, using `Set-Location` or `cd` to return to a PSDrive sent users to the default location for that drive. Users are now sent to the last known current working directory for that session.

`cd -` returns to previous directory

PowerShell

```
C:\Windows\System32> cd C:\  
C:\> cd -  
C:\Windows\System32>
```

Or on Linux:

ShellSession

```
PS /etc> cd /usr/bin  
PS /usr/bin> cd -  
PS /etc>
```

Also, `cd` and `cd --` change to `$HOME`.

`Update-Help` as non-admin

By popular demand, `Update-Help` no longer needs to be run as an administrator.

`Update-Help` now defaults to saving help to a user-scoped folder.

`Where-Object -Not`

With the addition of `-Not` parameter to `Where-Object`, can filter an object at the pipeline for the non-existence of a property, or a null/empty property value.

For example, this command returns all services that don't have any dependent services defined:

PowerShell

```
Get-Service | Where-Object -Not DependentServices
```

Changes to Web Cmdlets

The underlying .NET API of the Web Cmdlets has been changed to `System.Net.Http.HttpClient`. This change provides many benefits. However, this change along with a lack of interoperability with Internet Explorer have resulted in several breaking changes within `Invoke-WebRequest` and `Invoke-RestMethod`.

- `Invoke-WebRequest` now supports basic HTML Parsing only. `Invoke-WebRequest` always returns a `BasicHtmlWebResponseObject` object. The `ParsedHtml` and `Forms` properties have been removed.
- `BasicHtmlWebResponseObject.Headers` values are now `String[]` instead of `String`.
- `BasicHtmlWebResponseObject.BaseResponse` is now a `System.Net.Http.HttpResponseMessage` object.
- The `Response` property on Web Cmdlet exceptions is now a `System.Net.Http.HttpResponseMessage` object.
- Strict RFC header parsing is now default for the `-Headers` and `-UserAgent` parameter. This can be bypassed with `-SkipHeaderValidation`.
- `file://` and `ftp://` URI schemes are no longer supported.
- `System.Net.ServicePointManager` settings are no longer honored.
- There is currently no certificate based authentication available on macOS.
- Use of `-Credential` over an `http://` URI will result in an error. Use an `https://` URI or supply the `-AllowUnencryptedAuthentication` parameter to suppress the error.
- `-MaximumRedirection` now produces a terminating error when redirection attempts exceed the provided limit instead of returning the results of the last redirection.
- In PowerShell 6.2, a change was made to default to UTF-8 encoding for JSON responses. When a charset is not supplied for a JSON response, the default encoding should be UTF-8 per RFC 8259.
- Default encoding set to UTF-8 for `application-json` responses
- Added `-SkipHeaderValidation` parameter to allow `Content-Type` headers that aren't standards-compliant
- Added `-Form` parameter to support simplified `multipart/form-data` support
- Compliant, case-insensitive handling of relation keys
- Added `-Resume` parameter for web cmdlets

Invoke-RestMethod returns useful info when no data is returned

When an API returns just `null`, `Invoke-RestMethod` was serializing this as the string `"null"` instead of `$null`. This change fixes the logic in `Invoke-RestMethod` to properly serialize a valid single value JSON `null` literal as `$null`.

Web Cmdlets warn when `-Credential` is sent over unencrypted connections

When using HTTP, content including passwords are sent as clear-text. This change is to not allow this by default and return an error if credentials are being passed insecurely. Users can bypass this by using the `-AllowUnencryptedAuthentication` switch.

Make `-OutFile` parameter in web cmdlets to work like `-LiteralPath`

Beginning in PowerShell 7.1, the `OutFile` parameter of the web cmdlets works like `LiteralPath` and does not process wildcards.

API changes

Remove `AddTypeCommandBase` class

The `AddTypeCommandBase` class was removed from `Add-Type` to improve performance. This class is only used by the `Add-Type` cmdlet and should not impact users.

Removed `VisualBasic` as a supported language in `Add-Type`

In the past, you could compile Visual Basic code using the `Add-Type` cmdlet. Visual Basic was rarely used with `Add-Type`. We removed this feature to reduce the size of PowerShell.

Removed `RunspaceConfiguration` support

Previously, when creating a PowerShell runspace programmatically using the API, you could use the legacy `RunspaceConfiguration` or the newer `InitialSessionState` classes. This change removed support for `RunspaceConfiguration` and only supports `InitialSessionState`.

`CommandInvocationIntrinsics.InvokeScript` bind arguments to `$input` instead of `$args`

An incorrect position of a parameter resulted in the args passed as input instead of as args.

Remove `ClrVersion` and `BuildVersion` properties from `$PSVersionTable`

The `ClrVersion` property of `$PSVersionTable` is not useful with CoreCLR. End users should not be using that value to determine compatibility.

The `BuildVersion` property was tied to the Windows build version, which is not available on non-Windows platforms. Use the `GitCommitId` property to retrieve the exact build version of PowerShell.

Implement Unicode escape parsing

`\u####` or `\u{####}` is converted to the corresponding Unicode character. To output a literal `\u`, escape the backtick: `\`u`.

Parameter binding problem with `ValueFromRemainingArguments` in PS functions

`ValueFromRemainingArguments` now returns the values as an array instead of a single value which itself is an array.

Cleaned up uses of `CommandTypes.Workflow` and `WorkflowInfoCleaned`

Clean up code related to the uses of `CommandTypes.Workflow` and `WorkflowInfo` in `System.Management.Automation`.

These minor breaking changes mainly affect help provider code.

- Change the public constructors of `WorkflowInfo` to internal. We don't support workflow anymore, so it makes sense to not allow people to create `Workflow` instances.
- Remove the type `System.Management.Automation.DebugSource` since it's only used for workflow debugging.
- Remove the overload of `SetParent` from the abstract class `Debugger` that is only used for workflow debugging.
- Remove the same overload of `SetParent` from the derived class `RemotingJobDebugger`.

Do not wrap return result in `PSObject` when converting a `ScriptBlock` to a delegate

When a `ScriptBlock` is converted to a delegate type to be used in C# context, wrapping the result in a `PSObject` brings unneeded troubles:

- When the value is converted to the delegate return type, the `PSObject` essentially gets unwrapped. So the `PSObject` is unneeded.
- When the delegate return type is `object`, it gets wrapped in a `PSObject` making it hard to work with in C# code.

After this change, the returned object is the underlying object.

Remoting Support

PowerShell Remoting (PSRP) using WinRM on Unix platforms requires NTLM/Negotiate or Basic Auth over HTTPS. PSRP on macOS only supports Basic Auth over HTTPS. Kerberos-based authentication is not supported for non-Windows platforms.

PowerShell also supports PowerShell Remoting (PSRP) over SSH on all platforms (Windows, macOS, and Linux). For more information, see [SSH remoting in PowerShell](#).

PowerShell Direct for Containers tries to use `pwsh` first

[PowerShell Direct](#) is a feature of PowerShell and Hyper-V that allows you to connect to a Hyper-V VM or Container without network connectivity or other remote management services.

In the past, PowerShell Direct connected using the built-in Windows PowerShell instance on the Container. Now, PowerShell Direct first attempts to connect using any available `pwsh.exe` on the `PATH` environment variable. If `pwsh.exe` isn't available, PowerShell Direct falls back to use `powershell.exe`.

`Enable-PSRemoting` now creates separate remoting endpoints for preview versions

`Enable-PSRemoting` now creates two remoting session configurations:

- One for the major version of PowerShell. For example, `PowerShell.6`. This endpoint that can be relied upon across minor version updates as the "system-wide" PowerShell 6 session configuration

- One version-specific session configuration, for example: `PowerShell.6.1.0`

This behavior is useful if you want to have multiple PowerShell 6 versions installed and accessible on the same machine.

Additionally, preview versions of PowerShell now get their own remoting session configurations after running the `Enable-PSRemoting` cmdlet:

```
PowerShell  
C:\WINDOWS\system32> Enable-PSRemoting
```

Your output may be different if you haven't set up WinRM before.

```
Output  
WinRM is already set up to receive requests on this computer.  
WinRM is already set up for remote management on this computer.
```

Then you can see separate PowerShell session configurations for the preview and stable builds of PowerShell 6, and for each specific version.

```
PowerShell  
Get-PSSessionConfiguration  
  
Output  
Name : PowerShell.6.2-preview.1  
PSVersion : 6.2  
StartupScript :  
RunAsUser :  
Permission : NT AUTHORITY\INTERACTIVE AccessAllowed,  
BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management Users  
AccessAllowed  
  
Name : PowerShell.6-preview  
PSVersion : 6.2  
StartupScript :  
RunAsUser :  
Permission : NT AUTHORITY\INTERACTIVE AccessAllowed,  
BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management Users  
AccessAllowed  
  
Name : powershell.6  
PSVersion : 6.1  
StartupScript :  
RunAsUser :
```

```
Permission      : NT AUTHORITY\INTERACTIVE AccessAllowed,  
BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management Users  
AccessAllowed  
  
Name          : powershell.6.1.0  
PSVersion     : 6.1  
StartupScript :  
RunAsUser     :  
Permission    : NT AUTHORITY\INTERACTIVE AccessAllowed,  
BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management Users  
AccessAllowed
```

`user@host:port` syntax supported for SSH

SSH clients typically support a connection string in the format `user@host:port`. With the addition of SSH as a protocol for PowerShell Remoting, we've added support for this format of connection string:

```
Enter-PSSession -HostName fooUser@ssh.contoso.com:2222
```

Telemetry can only be disabled with an environment variable

PowerShell sends basic telemetry data to Microsoft when it is launched. The data includes the OS name, OS version, and PowerShell version. This data allows us to better understand the environments where PowerShell is used and enables us to prioritize new features and fixes.

To opt-out of this telemetry, set the environment variable `POWERSHELL_TELEMETRY_OPTOUT` to `true`, `yes`, or `1`. We no longer support deletion of the file

```
DELETE_ME_TO_DISABLE_CONSOLEHOST_TELEMETRY
```

 to disable telemetry.

PowerShell differences on non-Windows platforms

Article • 07/18/2024

PowerShell strives to provide feature parity across all supported platforms. However, some features behave differently or aren't available due to differences in .NET Core and platform-specific differences. Other changes were made to improve the interoperability of PowerShell on non-Windows platforms.

.NET Framework vs .NET Core

PowerShell on Linux and macOS uses .NET Core, a subset of the full .NET Framework on Microsoft Windows. As a result, scripts that run on Windows might not run on non-Windows platforms because of the differences in the frameworks.

For more information about changes in .NET Core, see [Breaking changes for migration from .NET Framework to .NET Core](#).

General Unix interoperability changes

- Added support for native command globbing on Unix platforms. This means you can use wildcards with native commands like `ls *.txt`.
- The `more` functionality respects the Linux `$PAGER` and defaults to `less`.
- Trailing backslash is automatically escaped when dealing with native command arguments.
- Fixed ConsoleHost to honor `NoEcho` on Unix platforms.
- Don't add `PATHEXT` environment variable on Unix.
- A `powershell` man-page is included in the package.

Execution policy

PowerShell ignores execution policies when running on non-Windows platforms. `Get-ExecutionPolicy` returns **Unrestricted** on Linux and macOS. `Set-ExecutionPolicy` does nothing on Linux and macOS.

Case-sensitivity in PowerShell

Historically, PowerShell has been uniformly case-insensitive, with few exceptions. On Unix-like operating systems, the file system is predominantly case-sensitive, and PowerShell adheres to the standard of the file system.

- You must use the correct case when a filename is specified in PowerShell.
- If a script tries to load a module and the module name isn't cased correctly, then the module load fails. This behavior might cause a problem with existing scripts if the name referenced by the module doesn't match the proper case of the actual filename.
- While names in the filesystem are case-sensitive, tab-completion of filenames isn't case-sensitive. Tab-completion cycles through the list of names using case-insensitive matching.
- `Get-Help` supports case-insensitive pattern matching on Unix platforms.
- `Import-Module` is case insensitive when used with a filename to determine the module name.

Filesystem support for Linux and macOS

- Paths given to cmdlets are now slash-agnostic (both `/` and `\` work as directory separators)
- XDG Base Directory Specification is now respected and used by default:
 - The Linux/macOS profile path is located at `~/.config/powershell/profile.ps1`
 - The history save path is located at
`~/.local/share/powershell/PSReadLine/ConsoleHost_history.txt`
 - The user module path is located at `~/.local/share/powershell/Modules`
- Support for file and folder names containing the colon character on Unix.
- Support for script names or full paths that have commas.
- Detect when the `LiteralPath` parameter is used to suppress wildcard expansion for navigation cmdlets.
- Updated `Get-ChildItem` to work more like the *nix `ls -R` and the Windows `DIR /S` native commands. `Get-ChildItem` now returns the symbolic links encountered during a recursive search and doesn't search the directories that those links target.

.PS1 File Extensions

PowerShell scripts must end in `.ps1` for the interpreter to understand how to load and run them in the current process. Running scripts in the current process is the expected usual behavior for PowerShell. You can add the `#!` magic number to a script that doesn't have a `.ps1` extension, but this causes the script to be run in a new PowerShell

instance, preventing the script from working correctly when interchanging objects. This behavior might be desirable when executing a PowerShell script from Bash or another shell.

Convenience aliases removed

PowerShell provides a set of aliases on Windows that map to Linux command names for user convenience. On Linux and macOS, the "convenience aliases" for the basic commands `ls`, `cp`, `mv`, `rm`, `cat`, `man`, `mount`, and `ps` were removed to allow the native executable to run without specifying a path.

Logging

On macOS, PowerShell uses the native `os_log` APIs to log to Apple's [unified logging system](#). On Linux, PowerShell uses [Syslog](#), a ubiquitous logging solution.

Job Control

There's no Unix-style job-control support in PowerShell on Linux or macOS. The `fg` and `bg` commands aren't available. However, you can use [PowerShell jobs](#) that work on all platforms.

Putting `&` at the end of a pipeline causes the pipeline to be run as a PowerShell job. When a pipeline is backgrounded, a job object is returned. Once the pipeline is running as a job, all `*-Job` cmdlets can be used to manage the job. Variables (ignoring process-specific variables) used in the pipeline are automatically copied to the job so `Copy-Item $foo $bar &` just works. The job runs in the current directory instead of the user's home directory.

Remoting Support

PowerShell Remoting (PSRP) using WinRM on Unix platforms requires NTLM/Negotiate or Basic Auth over HTTPS. PSRP on macOS only supports Basic Auth over HTTPS. Kerberos-based authentication isn't supported.

PowerShell supports PowerShell Remoting (PSRP) over SSH on all platforms (Windows, Linux, and macOS). For more information, see [SSH remoting in PowerShell](#).

Just-Enough-Administration (JEA) Support

PowerShell on Linux or macOS doesn't allow you to create constrained administration (JEA) remoting endpoints.

`sudo`, `exec`, and PowerShell

Because PowerShell runs most commands in memory (like Python or Ruby), you can't use `sudo` directly with PowerShell built-ins. You can run `pwsh` from `sudo`. If it's necessary to run a PowerShell cmdlet from within PowerShell with `sudo`, for example, `sudo Set-Date 8/18/2016`, then you would use `sudo pwsh Set-Date 8/18/2016`.

Modules included on non-Windows platforms

For non-Windows platforms, PowerShell includes the following modules:

- Microsoft.PowerShell.Archive
- Microsoft.PowerShell.Core
- Microsoft.PowerShell.Host
- Microsoft.PowerShell.Management
- Microsoft.PowerShell.Security
- Microsoft.PowerShell.Utility
- PackageManagement
- PowerShellGet
- PSReadLine
- ThreadJob

A large number of the commands (cmdlets) commonly available in PowerShell aren't available on Linux or macOS. Often, these commands don't apply to these platforms. For example, commands for Windows-specific features like the registry or services aren't available. Other commands, like `Set-ExecutionPolicy`, are present but not functional.

For a comprehensive list of modules and cmdlets and the platforms they support, see [Release history of modules and cmdlets](#).

Modules no longer shipped with PowerShell

For various compatibility reasons, the following modules are no longer included in PowerShell.

- ISE
- Microsoft.PowerShell.LocalAccounts
- Microsoft.PowerShell.ODataUtils

- Microsoft.PowerShell.Operation.Validation
- PSScheduledJob
- PSWorkflow
- PSWorkflowUtility

The following Windows-specific modules aren't included in PowerShell for Linux or macOS.

- CimCmdlets
- Microsoft.PowerShell.Diagnostics
- Microsoft.WSMan.Management
- PSDiagnostics

Cmdlets not available on non-Windows platforms

Some cmdlets were removed from PowerShell. Others aren't available or might work differently on non-Windows platforms. For a comprehensive list of cmdlets removed from PowerShell, see [Cmdlets removed from PowerShell](#).

Microsoft.PowerShell.Core

The following cmdlets aren't available on Linux or macOS:

- `Disable-PSRemoting`
- `Enable-PSRemoting`
- `Connect-PSSession`
- `Disconnect-PSSession`
- `Receive-PSSession`
- `Get-PSSessionCapability`
- `Disable-PSSessionConfiguration`
- `Enable-PSSessionConfiguration`
- `Get-PSSessionConfiguration`
- `Register-PSSessionConfiguration`
- `Set-PSSessionConfiguration`
- `Unregister-PSSessionConfiguration`
- `Test-PSSessionConfigurationFile`

The `ShowWindow` parameter of `Get-Help` isn't available for non-Windows platforms. PowerShell 7.3 added the `Switch-Process` cmdlet and the `exec` function for Linux and

macOS. These commands aren't available on Windows.

Microsoft.PowerShell.Security cmdlets

The following cmdlets aren't available on Linux or macOS:

- `Get-Acl`
- `Set-Acl`
- `Get-AuthenticodeSignature`
- `Set-AuthenticodeSignature`
- `New-FileCatalog`
- `Test-FileCatalog`

These cmdlets are only available beginning in PowerShell 7.1.

- `Get-CmsMessage`
- `Protect-CmsMessage`
- `Unprotect-CmsMessage`

Microsoft.PowerShell.Management cmdlets

The following cmdlets aren't available on Linux and macOS:

- `Rename-Computer`
- `Get-ComputerInfo`
- `Get-HotFix`
- `Clear-RecycleBin`
- `Get-Service`
- `New-Service`
- `Remove-Service`
- `Restart-Service`
- `Resume-Service`
- `Set-Service`
- `Start-Service`
- `Stop-Service`
- `Suspend-Service`
- `Set-TimeZone`

The following cmdlets are available with limitations:

- `Get-Clipboard` - available in PowerShell 7.0+

- `Set-Clipboard` - available in PowerShell 7.0+
- `Restart-Computer` - available for Linux and macOS in PowerShell 7.1+
- `Stop-Computer` - available for Linux and macOS in PowerShell 7.1+

Microsoft.PowerShell.Utility cmdlets

The following cmdlets aren't available on Linux and macOS:

- `Convert-String`
- `ConvertFrom-String`
- `ConvertFrom-SddlString`
- `Out-GridView`
- `Out-Printer`
- `Show-Command`

Aliases not available on Linux or macOS

The following table lists the aliases available for Windows that aren't available on non-Windows platforms. These aliases aren't available because the alias conflicts with a native command on those platforms.

[] Expand table

Alias	Cmdlet
<code>ac</code>	<code>Add-Content</code>
<code>cat</code>	<code>Get-Content</code>
<code>clear</code>	<code>Clear-Host</code>
<code>compare</code>	<code>Compare-Object</code>
<code>cp</code>	<code>Copy-Item</code>
<code>cpp</code>	<code>Copy-ItemProperty</code>
<code>diff</code>	<code>Compare-Object</code>
<code>kill</code>	<code>Stop-Process</code>
<code>ls</code>	<code>Get-ChildItem</code>
<code>man</code>	<code>help</code>

Alias	Cmdlet
mount	New-PSDrive
mv	Move-Item
ps	Get-Process
rm	Remove-Item
rmdir	Remove-Item
sleep	Start-Sleep
sort	Sort-Object
start	Start-Process
tee	Tee-Object
write	Write-Output

The table doesn't include aliases unavailable for cmdlets that don't exist on non-Windows platforms.

PowerShell Desired State Configuration (DSC)

Beginning with PowerShell 7.2, the `PSDesiredStateConfiguration` module was removed from PowerShell and is published in the PowerShell Gallery. For more information, see the [announcement](#) on the PowerShell Team blog. For more information about using DSC on Linux, see [Get started with DSC for Linux](#). DSC v1.1 and v2.x aren't supported on macOS. DSC v3 is supported on Windows, Linux, and macOS, but it's still in early development.

Release history of modules and cmdlets

Article • 01/17/2025

This article lists the modules and cmdlets that are included in various versions of PowerShell. This is a summary of information found in the release notes. More detailed information can be found in the release notes:

- [What's new in PowerShell 7.5](#)
- [What's new in PowerShell 7.4](#)
- [What's new in PowerShell 7.3](#)
- [What's new in PowerShell 7.2](#)
- [What's new in PowerShell 7.1](#)
- [What's new in PowerShell 7.0](#)

This is a work in progress. Please help us keep this information fresh.

Module release history

[+] [Expand table](#)

ModuleName / PSVersion	5.1	7.4	7.5	7.6	Note
CimCmdlets	✓	✓	✓	✓	Windows only
ISE (introduced in 2.0)	✓				Windows only
Microsoft.PowerShell.Archive	✓	✓	✓	✓	
Microsoft.PowerShell.Core	✓	✓	✓	✓	
Microsoft.PowerShell.Diagnostics	✓	✓	✓	✓	Windows only
Microsoft.PowerShell.Host	✓	✓	✓	✓	
Microsoft.PowerShell.LocalAccounts	✓				Windows only (64-bit only)
Microsoft.PowerShell.Management	✓	✓	✓	✓	
Microsoft.PowerShell.ODataUtils	✓				Windows only
Microsoft.PowerShell.Operation.Validation	✓				Windows only
Microsoft.PowerShell.PSResourceGet	✓	✓	✓	✓	New versions available from the Gallery

ModuleName / PSVersion	5.1	7.4	7.5	7.6	Note
Microsoft.PowerShell.Security	✓	✓	✓	✓	
Microsoft.PowerShell.Utility	✓	✓	✓	✓	
Microsoft.WsMan.Management	✓	✓	✓	✓	Windows only
PackageManagement	✓	✓	✓	✓	
PowerShellGet 1.1	✓				Must upgrade to v2.x
PowerShellGet 2.x		✓	✓	✓	New versions available from the Gallery
PSDesiredStateConfiguration 1.1	✓				Removed in 7.2 - available from the Gallery
PSDesiredStateConfiguration 2.x					Removed in 7.2 - available from the Gallery
PSDesiredStateConfiguration 3.x					Preview available from the Gallery
PSDiagnostics	✓	✓	✓	✓	Windows only
PSReadLine	v1.x	v2.3.4	v2.3.4	v2.3.6	New versions available from the Gallery
PSScheduledJob	✓				Windows only
PSWorkflow	✓				Windows only
PSWorkflowUtility	✓				Windows only
ThreadJob	✓	✓	✓		Can be installed in PowerShell 5.1

Cmdlet release history

CimCmdlets

[] Expand table

Cmdlet name	5.1	7.4	7.5	7.6	Note
Export-BinaryMiLog	✓				Windows only
Get-CimAssociatedInstance	✓	✓	✓	✓	Windows only
Get-CimClass	✓	✓	✓	✓	Windows only
Get-CimInstance	✓	✓	✓	✓	Windows only
Get-CimSession	✓	✓	✓	✓	Windows only
Import-BinaryMiLog	✓				Windows only
Invoke-CimMethod	✓	✓	✓	✓	Windows only
New-CimInstance	✓	✓	✓	✓	Windows only
New-CimSession	✓	✓	✓	✓	Windows only
New-CimSessionOption	✓	✓	✓	✓	Windows only
Register-CimIndicationEvent	✓	✓	✓	✓	Windows only
Remove-CimInstance	✓	✓	✓	✓	Windows only
Remove-CimSession	✓	✓	✓	✓	Windows only
Set-CimInstance	✓	✓	✓	✓	Windows only

ISE (introduced in 2.0)

This module is only available in Windows PowerShell.

[\[+\] Expand table](#)

Cmdlet name	5.1	Note
Get-IseSnippet	✓	
Import-IseSnippet	✓	
New-IseSnippet	✓	

Microsoft.PowerShell.Archive

[\[+\] Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
Compress-Archive	✓	✓	✓	✓	
Expand-Archive	✓	✓	✓	✓	

Microsoft.PowerShell.Core

[\[+\] Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
Add-History	✓	✓	✓	✓	
Add-PSSnapin	✓				Windows only
Clear-History	✓	✓	✓	✓	
Clear-Host	✓	✓	✓	✓	
Connect-PSSession	✓	✓	✓	✓	Windows only
Debug-Job	✓	✓	✓	✓	
Disable-ExperimentalFeature	✓	✓	✓	✓	Added in 6.2
Disable-PSRemoting	✓	✓	✓	✓	Windows only
Disable-PSSessionConfiguration	✓	✓	✓	✓	Windows only
Disconnect-PSSession	✓	✓	✓	✓	Windows only
Enable-ExperimentalFeature	✓	✓	✓	✓	Added in 6.2
Enable-PSRemoting	✓	✓	✓	✓	Windows only
Enable-PSSessionConfiguration	✓	✓	✓	✓	Windows only
Enter-PHostProcess	✓	✓	✓	✓	Added Linux support in 6.2
Enter-PSSession	✓	✓	✓	✓	
Exit-PHostProcess	✓	✓	✓	✓	Added Linux support in 6.2
Exit-PSSession	✓	✓	✓	✓	
Export-Console	✓				Windows only
Export-ModuleMember	✓	✓	✓	✓	
ForEach-Object	✓	✓	✓	✓	

Cmdlet name	5.1	7.4	7.5	7.6	Note
Get-Command	✓	✓	✓	✓	
Get-ExperimentalFeature		✓	✓	✓	Added in 6.2
Get-Help	✓	✓	✓	✓	
Get-History	✓	✓	✓	✓	
Get-Job	✓	✓	✓	✓	
Get-Module	✓	✓	✓	✓	
Get-PSHostProcessInfo	✓	✓	✓	✓	Added Linux support in 6.2
Get-PSSession	✓	✓	✓	✓	
Get-PSSessionCapability	✓	✓	✓	✓	
Get-PSSessionConfiguration	✓	✓	✓	✓	
Get-PSSnapin	✓				Windows only
Get-Verb	✓				Moved to Microsoft.PowerShell.Utility 6.0+
Import-Module	✓	✓	✓	✓	
Invoke-Command	✓	✓	✓	✓	
Invoke-History	✓	✓	✓	✓	
New-Module	✓	✓	✓	✓	
New-ModuleManifest	✓	✓	✓	✓	
New-PSRoleCapabilityFile	✓	✓	✓	✓	
New-PSSession	✓	✓	✓	✓	
New-PSSessionConfigurationFile	✓	✓	✓	✓	Added Linux support in 7.3
New-PSSessionOption	✓	✓	✓	✓	
New-PSTransportOption	✓	✓	✓	✓	
Out-Default	✓	✓	✓	✓	
Out-Host	✓	✓	✓	✓	
Out-Null	✓	✓	✓	✓	
Receive-Job	✓	✓	✓	✓	

Cmdlet name	5.1	7.4	7.5	7.6	Note
Receive-PSSession	✓	✓	✓	✓	Windows only
Register-ArgumentCompleter	✓	✓	✓	✓	
Register-PSSessionConfiguration	✓	✓	✓	✓	Windows only
Remove-Job	✓	✓	✓	✓	
Remove-Module	✓	✓	✓	✓	
Remove-PSSession	✓	✓	✓	✓	
Remove-PSSnapin	✓				Windows only
Resume-Job	✓				
Save-Help	✓	✓	✓	✓	
Set-PSDebug	✓	✓	✓	✓	
Set-PSSessionConfiguration	✓	✓	✓	✓	Windows only
Set-StrictMode	✓	✓	✓	✓	
Start-Job	✓	✓	✓	✓	
Stop-Job	✓	✓	✓	✓	
Switch-Process		✓	✓		Linux and macOS only
Suspend-Job	✓				Windows only
Test-ModuleManifest	✓	✓	✓	✓	
Test-PSSessionConfigurationFile	✓	✓	✓	✓	Windows only
Unregister-PSSessionConfiguration	✓	✓	✓	✓	Windows only
Update-Help	✓	✓	✓	✓	
Wait-Job	✓	✓	✓	✓	
Where-Object	✓	✓	✓	✓	

Microsoft.PowerShell.Diagnostics

[\[+\] Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
Export-Counter	✓				Windows only
Get-Counter	✓	✓	✓	✓	Windows only
Get-WinEvent	✓	✓	✓	✓	Windows only
Import-Counter	✓				Windows only
New-WinEvent	✓	✓	✓	✓	Windows only

Microsoft.PowerShell.Host

[\[+\] Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
Start-Transcript	✓	✓	✓	✓	
Stop-Transcript	✓	✓	✓	✓	

Microsoft.PowerShell.LocalAccounts (64-bit only)

This module is only available in Windows PowerShell.

[\[+\] Expand table](#)

Cmdlet name	5.1	Note
Add-LocalGroupMember	✓	
Disable-LocalUser	✓	
Enable-LocalUser	✓	
Get-LocalGroup	✓	
Get-LocalGroupMember	✓	
Get-LocalUser	✓	
New-LocalGroup	✓	
New-LocalUser	✓	
Remove-LocalGroup	✓	

Cmdlet name	5.1	Note
Remove-LocalGroupMember	✓	
Remove-LocalUser	✓	
Rename-LocalGroup	✓	
Rename-LocalUser	✓	
Set-LocalGroup	✓	
Set-LocalUser	✓	

Microsoft.PowerShell.Management

[\[+\] Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
Add-Computer	✓				Windows only
Add-Content	✓	✓	✓	✓	
Checkpoint-Computer	✓				Windows only
Clear-Content	✓	✓	✓	✓	
Clear-EventLog	✓				Windows only
Clear-Item	✓	✓	✓	✓	
Clear-ItemProperty	✓	✓	✓	✓	
Clear-RecycleBin	✓	✓	✓	✓	Windows only
Complete-Transaction	✓				Windows only
Convert-Path	✓	✓	✓	✓	
Copy-Item	✓	✓	✓	✓	
Copy-ItemProperty	✓	✓	✓	✓	
Debug-Process	✓	✓	✓	✓	
Disable-ComputerRestore	✓				Windows only
Enable-ComputerRestore	✓				Windows only
Get-ChildItem	✓	✓	✓	✓	

Cmdlet name	5.1	7.4	7.5	7.6	Note
Get-Clipboard	✓	✓	✓	✓	
Get-ComputerInfo	✓	✓	✓	✓	Windows only
Get-ComputerRestorePoint	✓				Windows only
Get-Content	✓	✓	✓	✓	
Get-ControlPanelItem	✓				Windows only
Get-EventLog	✓				Windows only
Get-HotFix	✓	✓	✓	✓	Windows only
Get-Item	✓	✓	✓	✓	
Get-ItemProperty	✓	✓	✓	✓	
Get-ItemPropertyValue	✓	✓	✓	✓	
Get-Location	✓	✓	✓	✓	
Get-Process	✓	✓	✓	✓	
Get-PSDrive	✓	✓	✓	✓	
Get-PSPrinter	✓	✓	✓	✓	
Get-Service	✓	✓	✓	✓	Windows only
Get-TimeZone	✓	✓	✓	✓	Windows only
Get-Transaction	✓				Windows only
Get-WmiObject	✓				Windows only
Invoke-Item	✓	✓	✓	✓	
Invoke-WmiMethod	✓				Windows only
Join-Path	✓	✓	✓	✓	
Limit-EventLog	✓				Windows only
Move-Item	✓	✓	✓	✓	
Move-ItemProperty	✓	✓	✓	✓	
New-EventLog	✓				Windows only
New-Item	✓	✓	✓	✓	

Cmdlet name	5.1	7.4	7.5	7.6	Note
New-ItemProperty	✓	✓	✓	✓	
New-PSDrive	✓	✓	✓	✓	
New-Service	✓	✓	✓	✓	Windows only
New-WebServiceProxy	✓				Windows only
Pop-Location	✓	✓	✓	✓	
Push-Location	✓	✓	✓	✓	
Register-WmiEvent	✓				Windows only
Remove-Computer	✓				Windows only
Remove-EventLog	✓				Windows only
Remove-Item	✓	✓	✓	✓	
Remove-ItemProperty	✓	✓	✓	✓	
Remove-PSDrive	✓	✓	✓	✓	
Remove-Service		✓	✓	✓	Windows only
Remove-WmiObject	✓				Windows only
Rename-Computer	✓	✓	✓	✓	Windows only
Rename-Item	✓	✓	✓	✓	
Rename-ItemProperty	✓	✓	✓	✓	
Reset-ComputerMachinePassword	✓				Windows only
Resolve-Path	✓	✓	✓	✓	
Restart-Computer	✓	✓	✓	✓	Added Linux/macOS support in 7.1
Restart-Service	✓	✓	✓	✓	Windows only
Restore-Computer	✓				Windows only
Resume-Service	✓	✓	✓	✓	Windows only
Set-Clipboard	✓	✓	✓	✓	
Set-Content	✓	✓	✓	✓	
Set-Item	✓	✓	✓	✓	

Cmdlet name	5.1	7.4	7.5	7.6	Note
Set-ItemProperty	✓	✓	✓	✓	
Set-Location	✓	✓	✓	✓	
Set-Service	✓	✓	✓	✓	Windows only
Set-TimeZone	✓	✓	✓	✓	Windows only
Set-WmiInstance	✓				Windows only
Show-ControlPanelItem	✓				Windows only
Show-EventLog	✓				Windows only
Split-Path	✓	✓	✓	✓	
Start-Process	✓	✓	✓	✓	
Start-Service	✓	✓	✓	✓	Windows only
Start-Transaction	✓				Windows only
Stop-Computer	✓	✓	✓	✓	Added Linux/macOS support in 7.1
Stop-Process	✓	✓	✓	✓	
Stop-Service	✓	✓	✓	✓	Windows only
Suspend-Service	✓	✓	✓	✓	Windows only
Test-ComputerSecureChannel	✓				Windows only
Test-Connection	✓	✓	✓	✓	
Test-Path	✓	✓	✓	✓	
Undo-Transaction	✓				Windows only
Use-Transaction	✓				Windows only
Wait-Process	✓	✓	✓	✓	
Write-EventLog	✓				Windows only

Microsoft.PowerShell.ODataUtils

This module is only available in Windows PowerShell.

[+] Expand table

Cmdlet name	5.1	Note
Export-ODataEndpointProxy	✓	

Microsoft.PowerShell.Operation.Validation

This module is only available in Windows PowerShell.

[Expand table](#)

Cmdlet name	5.1	Note
Get-OperationValidation	✓	
Invoke-OperationValidation	✓	

Microsoft.PowerShell.PSResourceGet

[Expand table](#)

Cmdlet name	7.4	7.5	7.6	Note
Compress-PSResource	✓	✓	✓	Added in v1.1.0 of the module
Find-PSResource	✓	✓	✓	
Get-InstalledPSResource	✓	✓	✓	
Get-PSResource	✓	✓	✓	
Get-PSResourceRepository	✓	✓	✓	
Get-PSScriptFileInfo	✓	✓	✓	
Import-PSGetRepository	✓	✓	✓	
Install-PSResource	✓	✓	✓	
New-PSScriptFileInfo	✓	✓	✓	
Publish-PSResource	✓	✓	✓	
Register-PSResourceRepository	✓	✓	✓	
Save-PSResource	✓	✓	✓	
Set-PSResourceRepository	✓	✓	✓	

Cmdlet name	7.4	7.5	7.6	Note
Test-PSScriptFileInfo	✓	✓	✓	
Uninstall-PSResource	✓	✓	✓	
Unregister-PSResourceRepository	✓	✓	✓	
Update-PSModuleManifest	✓	✓	✓	
Update-PSResource	✓	✓	✓	
Update-PSScriptFileInfo	✓	✓	✓	

Microsoft.PowerShell.Security

[\[+\] Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
ConvertFrom-SecureString	✓	✓	✓	✓	
ConvertTo-SecureString	✓	✓	✓	✓	
Get-Acl	✓	✓	✓	✓	Windows only
Get-AuthenticodeSignature	✓	✓	✓	✓	Windows only
Get-CmsMessage	✓	✓	✓	✓	Support for Linux/macOS added in 7.1
Get-Credential	✓	✓	✓	✓	
Get-ExecutionPolicy	✓	✓	✓	✓	Returns Unrestricted on Linux/macOS
Get-PfxCertificate	✓	✓	✓	✓	
New-FileCatalog	✓	✓	✓	✓	Windows only
Protect-CmsMessage	✓	✓	✓	✓	Support for Linux/macOS added in 7.1
Set-Acl	✓	✓	✓	✓	Windows only
Set-AuthenticodeSignature	✓	✓	✓	✓	Windows only
Set-ExecutionPolicy	✓	✓	✓	✓	Does nothing on Linux/macOS
Test-FileCatalog	✓	✓	✓	✓	Windows only
Unprotect-CmsMessage	✓	✓	✓	✓	Support for Linux/macOS added in 7.1

Microsoft.PowerShell.Utility

[Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
Add-Member	✓	✓	✓	✓	
Add-Type	✓	✓	✓	✓	
Clear-Variable	✓	✓	✓	✓	
Compare-Object	✓	✓	✓	✓	
Convert-String	✓				
ConvertFrom-CliXml			✓	✓	Added in 7.5
ConvertFrom-Csv	✓	✓	✓	✓	
ConvertFrom-Json	✓	✓	✓	✓	
ConvertFrom-Markdown		✓	✓	✓	Added in 6.1
ConvertFrom-SddlString	✓	✓	✓	✓	Windows only
ConvertFrom-String	✓				
ConvertFrom-StringData	✓	✓	✓	✓	
ConvertTo-CliXml			✓	✓	Added in 7.5
ConvertTo-Csv	✓	✓	✓	✓	
ConvertTo-Html	✓	✓	✓	✓	
ConvertTo-Json	✓	✓	✓	✓	
ConvertTo-Xml	✓	✓	✓	✓	
Debug-Runspace	✓	✓	✓	✓	
Disable-PSBreakpoint	✓	✓	✓	✓	
Disable-RunspaceDebug	✓	✓	✓	✓	
Enable-PSBreakpoint	✓	✓	✓	✓	
Enable-RunspaceDebug	✓	✓	✓	✓	
Export-Alias	✓	✓	✓	✓	
Export-Clixml	✓	✓	✓	✓	

Cmdlet name	5.1	7.4	7.5	7.6	Note
Export-Csv	✓	✓	✓	✓	
Export-FormatData	✓	✓	✓	✓	
Export-PSSession	✓	✓	✓	✓	
Format-Custom	✓	✓	✓	✓	
Format-Hex	✓	✓	✓	✓	
Format-List	✓	✓	✓	✓	
Format-Table	✓	✓	✓	✓	
Format-Wide	✓	✓	✓	✓	
Get-Alias	✓	✓	✓	✓	
Get-Culture	✓	✓	✓	✓	
Get-Date	✓	✓	✓	✓	
Get-Error		✓	✓	✓	
Get-Event	✓	✓	✓	✓	No event sources available on Linux/macOS
Get-EventSubscriber	✓	✓	✓	✓	
Get-FileHash	✓	✓	✓	✓	
Get-FormatData	✓	✓	✓	✓	
Get-Host	✓	✓	✓	✓	
Get-MarkdownOption		✓	✓	✓	Added in 6.1
Get-Member	✓	✓	✓	✓	
Get-PSBreakpoint	✓	✓	✓	✓	
Get-PSCallStack	✓	✓	✓	✓	
Get-Random	✓	✓	✓	✓	
Get-Runspace	✓	✓	✓	✓	
Get-RunspaceDebug	✓	✓	✓	✓	
Get-SecureRandom	✓	✓	✓	✓	Added in 7.4
Get-TraceSource	✓	✓	✓	✓	

Cmdlet name	5.1	7.4	7.5	7.6	Note
Get-TypeData	✓	✓	✓	✓	
Get-UICulture	✓	✓	✓	✓	
Get-Unique	✓	✓	✓	✓	
Get-Uptime		✓	✓	✓	
Get-Variable	✓	✓	✓	✓	
Get-Verb		✓	✓	✓	Moved from Microsoft.PowerShell.Core
Group-Object	✓	✓	✓	✓	
Import-Alias	✓	✓	✓	✓	
Import-Clixml	✓	✓	✓	✓	
Import-Csv	✓	✓	✓	✓	
Import-LocalizedData	✓	✓	✓	✓	
Import-PowerShellDataFile	✓	✓	✓	✓	
Import-PSSession	✓	✓	✓	✓	
Invoke-Expression	✓	✓	✓	✓	
Invoke-RestMethod	✓	✓	✓	✓	
Invoke-WebRequest	✓	✓	✓	✓	
Join-String		✓	✓	✓	
Measure-Command	✓	✓	✓	✓	
Measure-Object	✓	✓	✓	✓	
New-Alias	✓	✓	✓	✓	
New-Event	✓	✓	✓	✓	No event sources available on Linux/macOS
New-Guid	✓	✓	✓	✓	
New-Object	✓	✓	✓	✓	
New-TemporaryFile	✓	✓	✓	✓	
New-TimeSpan	✓	✓	✓	✓	
New-Variable	✓	✓	✓	✓	

Cmdlet name	5.1	7.4	7.5	7.6	Note
Out-File	✓	✓	✓	✓	
Out-GridView	✓	✓	✓	✓	Windows only
Out-Printer	✓	✓	✓	✓	Windows only
Out-String	✓	✓	✓	✓	
Read-Host	✓	✓	✓	✓	
Register-EngineEvent	✓	✓	✓	✓	No event sources available on Linux/macOS
Register-ObjectEvent	✓	✓	✓	✓	
Remove-Alias		✓	✓	✓	
Remove-Event	✓	✓	✓	✓	No event sources available on Linux/macOS
Remove-PSBreakpoint	✓	✓	✓	✓	
Remove-TypeData	✓	✓	✓	✓	
Remove-Variable	✓	✓	✓	✓	
Select-Object	✓	✓	✓	✓	
Select-String	✓	✓	✓	✓	
Select-Xml	✓	✓	✓	✓	
Send-MailMessage	✓	✓	✓	✓	
Set-Alias	✓	✓	✓	✓	
Set-Date	✓	✓	✓	✓	
Set-MarkdownOption		✓	✓	✓	Added in 6.1
Set-PSBreakpoint	✓	✓	✓	✓	
Set-TraceSource	✓	✓	✓	✓	
Set-Variable	✓	✓	✓	✓	
Show-Command	✓	✓	✓	✓	Windows only
Show-Markdown		✓	✓	✓	Added in 6.1
Sort-Object	✓	✓	✓	✓	
Start-Sleep	✓	✓	✓	✓	

Cmdlet name	5.1	7.4	7.5	7.6	Note
Tee-Object	✓	✓	✓	✓	
Test-Json		✓	✓	✓	
Trace-Command	✓	✓	✓	✓	
Unblock-File	✓	✓	✓	✓	Added support for macOS in 7.0
Unregister-Event	✓	✓	✓	✓	No event sources available on Linux/macOS
Update-FormatData	✓	✓	✓	✓	
Update-List	✓	✓	✓	✓	
Update-TypeData	✓	✓	✓	✓	
Wait-Debugger	✓	✓	✓	✓	
Wait-Event	✓	✓	✓	✓	
Write-Debug	✓	✓	✓	✓	
Write-Error	✓	✓	✓	✓	
Write-Host	✓	✓	✓	✓	
Write-Information	✓	✓	✓	✓	
Write-Output	✓	✓	✓	✓	
Write-Progress	✓	✓	✓	✓	
Write-Verbose	✓	✓	✓	✓	
Write-Warning	✓	✓	✓	✓	

Microsoft.WsMan.Management

[\[+\] Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
Connect-WSMan	✓	✓	✓	✓	Windows only
Disable-WSManCredSSP	✓	✓	✓	✓	Windows only
Disconnect-WSMan	✓	✓	✓	✓	Windows only
Enable-WSManCredSSP	✓	✓	✓	✓	Windows only

Cmdlet name	5.1	7.4	7.5	7.6	Note
Get-WSManCredSSP	✓	✓	✓	✓	Windows only
Get-WSManInstance	✓	✓	✓	✓	Windows only
Invoke-WSManAction	✓	✓	✓	✓	Windows only
New-WSManInstance	✓	✓	✓	✓	Windows only
New-WSManSessionOption	✓	✓	✓	✓	Windows only
Remove-WSManInstance	✓	✓	✓	✓	Windows only
Set-WSManInstance	✓	✓	✓	✓	Windows only
Set-WSManQuickConfig	✓	✓	✓	✓	Windows only
Test-WSMan	✓	✓	✓	✓	Windows only

PackageManagement

[\[+\] Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
Find-Package	✓	✓	✓	✓	
Find-PackageProvider	✓	✓	✓	✓	
Get-Package	✓	✓	✓	✓	
Get-PackageProvider	✓	✓	✓	✓	
Get-PackageSource	✓	✓	✓	✓	
Import-PackageProvider	✓	✓	✓	✓	
Install-Package	✓	✓	✓	✓	
Install-PackageProvider	✓	✓	✓	✓	
Register-PackageSource	✓	✓	✓	✓	
Save-Package	✓	✓	✓	✓	
Set-PackageSource	✓	✓	✓	✓	
Uninstall-Package	✓	✓	✓	✓	
Unregister-PackageSource	✓	✓	✓	✓	

PowerShellGet 2.x

[Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
Find-Command	✓	✓	✓	✓	
Find-DscResource	✓	✓	✓	✓	
Find-Module	✓	✓	✓	✓	
Find-RoleCapability	✓	✓	✓	✓	
Find-Script	✓	✓	✓	✓	
Get-CredsFromCredentialProvider		✓	✓	✓	
Get-InstalledModule	✓	✓	✓	✓	
Get-InstalledScript	✓	✓	✓	✓	
Get-PSRepository	✓	✓	✓	✓	
Install-Module	✓	✓	✓	✓	
Install-Script	✓	✓	✓	✓	
New-ScriptFileInfo	✓	✓	✓	✓	
Publish-Module	✓	✓	✓	✓	
Publish-Script	✓	✓	✓	✓	
Register-PSRepository	✓	✓	✓	✓	
Save-Module	✓	✓	✓	✓	
Save-Script	✓	✓	✓	✓	
Set-PSRepository	✓	✓	✓	✓	
Test-ScriptFileInfo	✓	✓	✓	✓	
Uninstall-Module	✓	✓	✓	✓	
Uninstall-Script	✓	✓	✓	✓	
Unregister-PSRepository	✓	✓	✓	✓	
Update-Module	✓	✓	✓	✓	
Update-ModuleManifest	✓	✓	✓	✓	

Cmdlet name	5.1	7.4	7.5	7.6	Note
Update-Script	✓	✓	✓	✓	
Update-ScriptFileInfo	✓	✓	✓	✓	

PSDesiredStateConfiguration v1.1

This module is only available from in Windows PowerShell.

[\[+\] Expand table](#)

Cmdlet name	5.1	Note
Configuration	✓	
Disable-DscDebug	✓	
Enable-DscDebug	✓	
Get-DscConfiguration	✓	
Get-DscConfigurationStatus	✓	
Get-DscLocalConfigurationManager	✓	
Get-DscResource	✓	
Invoke-DscResource	✓	
New-DSCCheckSum	✓	
Publish-DscConfiguration	✓	
Remove-DscConfigurationDocument	✓	
Restore-DscConfiguration	✓	
Set-DscLocalConfigurationManager	✓	
Start-DscConfiguration	✓	
Stop-DscConfiguration	✓	
Test-DscConfiguration	✓	
Update-DscConfiguration	✓	

PSDesiredStateConfiguration v2.0.5

This module is only available from the PowerShell Gallery.

[+] Expand table

Cmdlet name	2.0.5	Note
Configuration	✓	
Get-DscResource	✓	
Invoke-DscResource	✓	Experimental
New-DSCChecksum	✓	

PSDesiredStateConfiguration v3.x - Preview

This module is only available from the PowerShell Gallery.

[+] Expand table

Cmdlet name	3.0 (preview)	Note
Configuration	✓	
ConvertTo-DscJsonSchema	✓	
Get-DscResource	✓	
Invoke-DscResource	✓	
New-DscChecksum	✓	

PSDiagnostics

[+] Expand table

Cmdlet name	5.1	7.4	7.5	7.6	Note
Disable-PSTrace	✓	✓	✓	✓	Windows only
Disable-PSWSManCombinedTrace	✓	✓	✓	✓	Windows only
Disable-WSManTrace	✓	✓	✓	✓	Windows only
Enable-PSTrace	✓	✓	✓	✓	Windows only
Enable-PSWSManCombinedTrace	✓	✓	✓	✓	Windows only

Cmdlet name	5.1	7.4	7.5	7.6	Note
Enable-WSManTrace	✓	✓	✓	✓	Windows only
Get-LogProperties	✓	✓	✓	✓	Windows only
Set-LogProperties	✓	✓	✓	✓	Windows only
Start-Trace	✓	✓	✓	✓	Windows only
Stop-Trace	✓	✓	✓	✓	Windows only

PSReadLine

[\[+\] Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
Get-PSReadLineKeyHandler	✓	✓	✓	✓	
Get-PSReadLineOption	✓	✓	✓	✓	
PSConsoleHostReadLine	✓	✓	✓	✓	
Remove-PSReadLineKeyHandler	✓	✓	✓	✓	
Set-PSReadLineKeyHandler	✓	✓	✓	✓	
Set-PSReadLineOption	✓	✓	✓	✓	

PSScheduledJob

This module is only available in Windows PowerShell.

[\[+\] Expand table](#)

Cmdlet name	5.1	Note
Add-JobTrigger	✓	
Disable-JobTrigger	✓	
Disable-ScheduledJob	✓	
Enable-JobTrigger	✓	
Enable-ScheduledJob	✓	

Cmdlet name	5.1	Note
Get-JobTrigger	✓	
Get-ScheduledJob	✓	
Get-ScheduledJobOption	✓	
New-JobTrigger	✓	
New-ScheduledJobOption	✓	
Register-ScheduledJob	✓	
Remove-JobTrigger	✓	
Set-JobTrigger	✓	
Set-ScheduledJob	✓	
Set-ScheduledJobOption	✓	
Unregister-ScheduledJob	✓	

PSWorkflow & PSWorkflowUtility

This module is only available in Windows PowerShell.

[\[+\] Expand table](#)

Cmdlet name	5.1	Note
New-PSWorkflowExecutionOption	✓	
New-PSWorkflowSession	✓	
Invoke-AsWorkflow	✓	

ThreadJob

[\[+\] Expand table](#)

Cmdlet name	5.1	7.4	7.5	7.6	Note
Start-ThreadJob	✓	✓	✓		Can be installed in PowerShell 5.1

PowerShell 7 module compatibility

Article • 06/28/2023

This article contains a partial list of PowerShell modules published by Microsoft.

The PowerShell team is working with the various feature teams that create PowerShell modules to help them produce modules that work in PowerShell 7. These modules are not owned by the PowerShell team.

The following modules are known to support PowerShell 7.

Azure PowerShell

The Az PowerShell module is a set of cmdlets for managing Azure resources directly from PowerShell. PowerShell 7.0.6 LTS or higher is the recommended version of PowerShell for use with the Azure Az PowerShell module on all platforms.

For more information, see [Introducing the Azure Az PowerShell module](#).

MSGraph PowerShell SDK

The Microsoft Graph SDKs are designed to simplify building high-quality, efficient, and resilient applications that access Microsoft Graph. PowerShell 7 and later is the recommended PowerShell version for use with the Microsoft Graph PowerShell SDK.

For more information, see [Install the Microsoft Graph PowerShell SDK](#).

Windows management modules

The Windows management modules provide management and support for various Windows features and services. Most of these modules have been updated to work natively with PowerShell 7, or tested for compatibility with PowerShell 7.

These modules are installed in different ways depending on the Edition of Windows, and how the module is packaged for that Edition.

For more information about installation and compatibility, see [PowerShell 7 module compatibility](#) in the Windows documentation.

Exchange Online Management 2.0

The Exchange Online PowerShell V2 module (EXO V2) connects to all Exchange-related PowerShell environments in Microsoft 365: Exchange Online PowerShell, Security & Compliance PowerShell, and standalone Exchange Online Protection (EOP) PowerShell.

EXO v2.0.4 or later is supported in PowerShell 7.0.3 or later.

For more information, see [About the Exchange Online PowerShell V2 module](#).

PowerShell modules for SQL Server

There are two SQL Server PowerShell modules:

- **SqlServer:** This module includes new cmdlets to support the latest SQL features, including updated versions of the cmdlets in SQLPS.
- **SQLPS:** The SQLPS is the module used by SQL Agent to run agent jobs in agent job steps using the PowerShell subsystem.

The SqlServer modules require PowerShell version 5.0 or greater.

For more information, see [Install the SQL Server PowerShell module](#).

Finding the status of other modules

You can find a complete list of modules using the [PowerShell Module Browser](#). Using the Module Browser, you can find documentation for other PowerShell modules to determine their PowerShell version requirements.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

The Windows PowerShell ISE

Article • 03/27/2025

The Windows PowerShell Integrated Scripting Environment (ISE) is a host application for Windows PowerShell. In the ISE, you can run commands and write, test, and debug scripts in a single Windows-based graphic user interface. The ISE provides multiline editing, tab completion, syntax coloring, selective execution, context-sensitive help, and support for right-to-left languages. Menu items and keyboard shortcuts are mapped to many of the same tasks that you would do in the Windows PowerShell console. For example, when you debug a script in the ISE, you can right-click on a line of code in the edit pane to set a breakpoint.

Support

The ISE was first introduced with Windows PowerShell V2 and was re-designed with PowerShell V3. The ISE is supported in all supported versions of Windows PowerShell up to and including Windows PowerShell V5.1.

Note

The PowerShell ISE is no longer in active feature development. As a shipping component of Windows, it continues to be officially supported for security and high-priority servicing fixes. We currently have no plans to remove the ISE from Windows.

There is no support for the ISE in PowerShell v6 and beyond. Users looking for replacement for the ISE should use [Visual Studio Code](#) with the [PowerShell Extension](#).

Key Features

Key features in Windows PowerShell ISE include:

- Multiline editing: To insert a blank line under the current line in the Command pane, press **SHIFT + ENTER**.
- Selective execution: To run part of a script, select the text you want to run, and then click the **Run Script** button. Or, press **F5**.
- Context-sensitive help: Type **Invoke-Item**, and then press **F1**. The Help file opens to the article for the **Invoke-Item** cmdlet.

The Windows PowerShell ISE lets you customize some aspects of its appearance. It also has its own Windows PowerShell profile script.

To start the Windows PowerShell ISE

Click **Start**, select **Windows PowerShell**, and then click **Windows PowerShell ISE**.

Alternately, you can type `powershell_ise.exe` in any command shell or in the Run box.

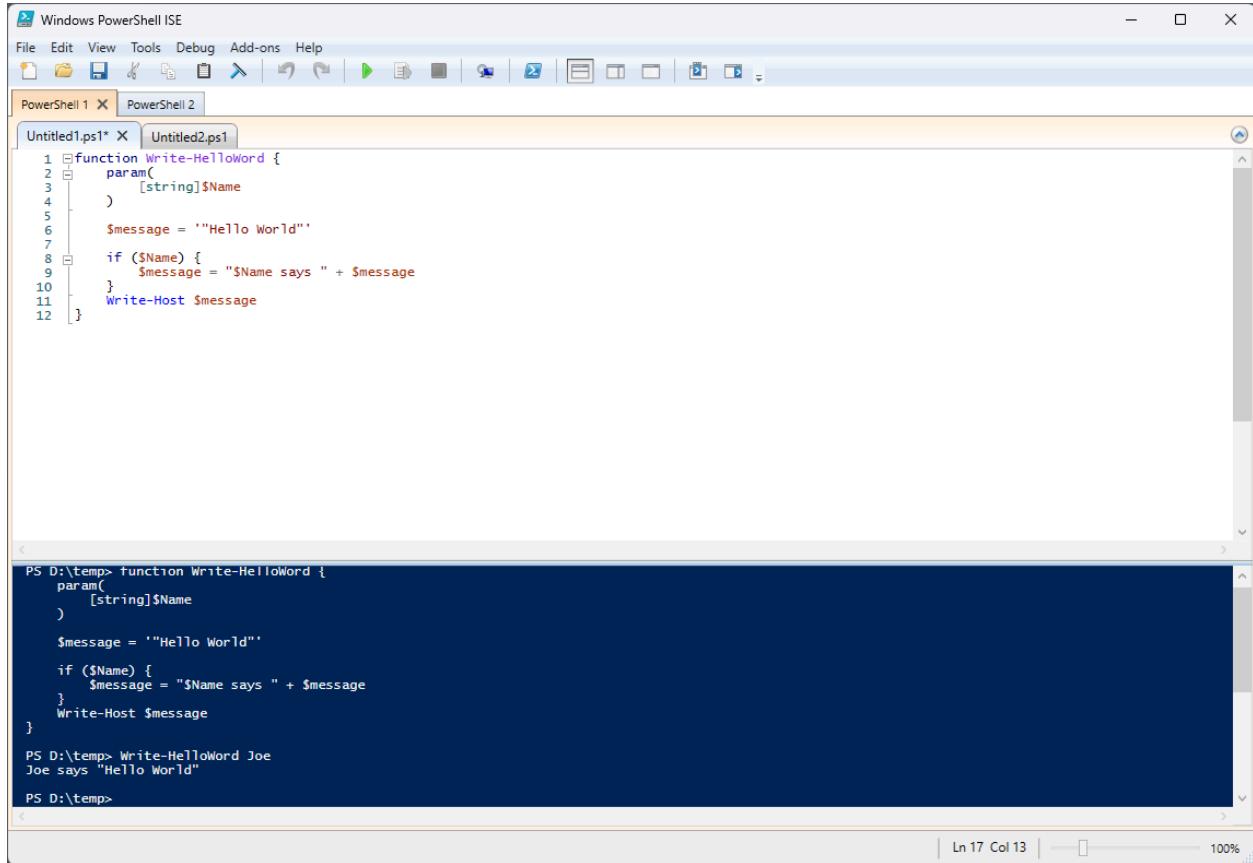
To get Help in the Windows PowerShell ISE

On the **Help** menu, click **Windows PowerShell Help**. Or, press `F1`. The file that opens describes Windows PowerShell ISE and Windows PowerShell, including all the help available from the `Get-Help` cmdlet.

Exploring the Windows PowerShell ISE

Article • 03/27/2025

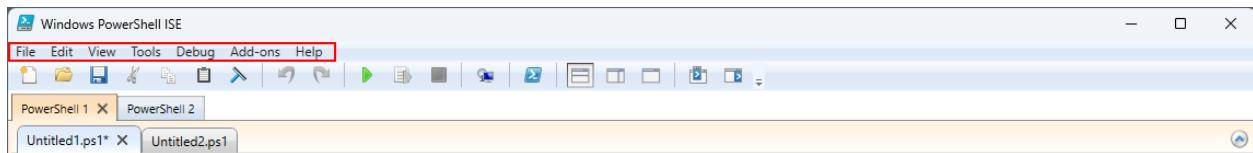
You can use the Windows PowerShell Integrated Scripting Environment (ISE) to create, run, and debug commands and scripts.



The Windows PowerShell ISE consists of the menu bar, Windows PowerShell tabs, the toolbar, script tabs, a Script Pane, a Console Pane, a status bar, a text-size slider and context-sensitive Help.

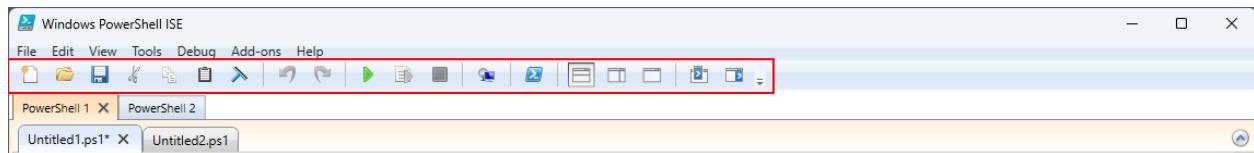
Menu Bar

The menu bar contains the **File**, **Edit**, **View**, **Tools**, **Debug**, **Add-ons**, and **Help** menus.



The buttons on the menus allow you to perform tasks related to writing and running scripts and running commands in the Windows PowerShell ISE. Additionally, an [add-on tool](#) may be placed on the menu bar by running scripts that use the [The ISE Object Model Hierarchy](#).

Toolbar



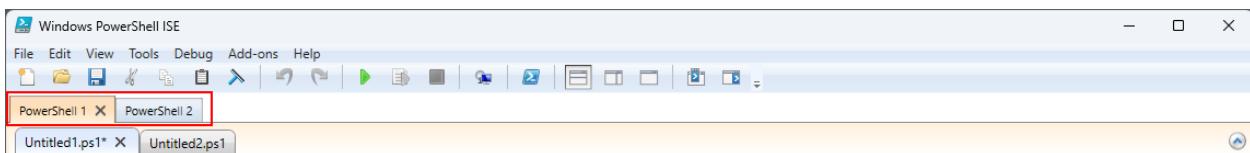
The following buttons are located on the toolbar.

[\[+\] Expand table](#)

Button	Function
New	Opens a new script.
Open	Opens an existing script or file.
Save	Saves a script or file.
Cut	Cuts the selected text and copies it to the clipboard.
Copy	Copies the selected text to the clipboard.
Paste	Pastes the contents of the clipboard at the cursor location.
Clear Console Pane	Clears all content in the Console Pane.
Undo	Reverses the action that was just performed.
Redo	Performs the action that was just undone.
Run Script	Runs a script.
Run Selection	Runs a selected portion of a script.
Stop Operation	Stops a script that's running.
New Remote PowerShell Tab	Creates a new PowerShell Tab that establishes a session on a remote computer. A dialog box appears and prompts you to enter details required to establish the remote connection.
Start powershell.exe	Opens a PowerShell Console.
Show Script Pane Top	Moves the Script Pane to the top in the display.
Show Script Pane Right	Moves the Script Pane to the right in the display.
Show Script Pane	Maximizes the Script Pane.

Button	Function
Maximized	
Show Command Window	Shows the Commands Pane for installed Modules, as a separate Window.
Show Command Add-on	Shows the Commands Pane for installed Modules, as a sidebar Add-on.

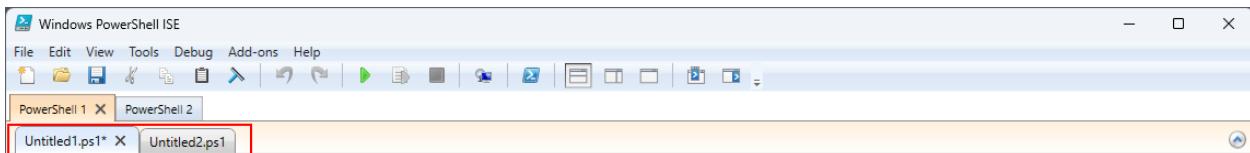
Windows PowerShell Tabs



A Windows PowerShell tab is the environment in which a Windows PowerShell script runs. You can open new Windows PowerShell tabs in the Windows PowerShell ISE to create separate environments on your local computer or on remote computers. You may have a maximum of eight PowerShell tabs simultaneously open.

For more information, see [How to Create a PowerShell Tab in Windows PowerShell ISE](#).

Script Tab



Displays the name of the script you are editing. You can click a script tab to select the script you want to edit.

When you point to the script tab, the fully qualified path to the script file appears in a tooltip.

Script Pane



Allows you to create and run scripts. You can open, edit and run existing scripts in the Script Pane. For more information, see [How to Write and Run Scripts in the Windows PowerShell ISE](#).

Console Pane

Displays the results of the commands and scripts you have run. You can run commands in the Console pane. You can also copy and clear the contents in the Console Pane.

For more information, see the following articles:

- [How to Use the Console Pane in the Windows PowerShell ISE](#)
- [How to Debug Scripts in Windows PowerShell ISE](#)
- [How to Use Tab Completion in the Script Pane and Console Pane](#)

Status Bar

Allows you to see whether the commands and scripts that you run are complete. The status bar is at the bottom of the window. Selected portions of error messages are displayed on the status bar.

Text-Size Slider

Increases or decreases the size of the text on the screen.

Help

Help for Windows PowerShell ISE is available on Microsoft Learn. You can open the Help by clicking **Windows PowerShell ISE Help** on the **Help** menu or by pressing the **F1** key anywhere except when the cursor is on a cmdlet name in either the Script Pane or the Console Pane. From the **Help** menu you can also run the `Update-Help` cmdlet, and display the Command Window, which assists you in constructing commands by showing you all the parameters for a cmdlet and enabling you to fill in the parameters in an easy-to-use form.

See Also

- [Introducing the Windows PowerShell ISE](#)
- [How to Use Profiles in Windows PowerShell ISE](#)
- [Accessibility in Windows PowerShell ISE](#)
- [Keyboard Shortcuts for the Windows PowerShell ISE](#)

How to Create a PowerShell Tab in Windows PowerShell ISE

Article • 03/27/2025

Tabs in the Windows PowerShell Integrated Scripting Environment (ISE) allow you to simultaneously create and use several execution environments within the same application. Each PowerShell tab corresponds to a separate execution environment or session.

ⓘ Note

Variables, functions, and aliases that you create in one tab don't carry over to another. They are different Windows PowerShell sessions.

Use the following steps to open or close a tab in Windows PowerShell. To rename a tab, set the [DisplayName](#) property on the Windows PowerShell Tab scripting object.

To create and use a new PowerShell Tab

On the **File** menu, click **New PowerShell Tab**. The new PowerShell tab always opens as the active window. PowerShell tabs are incrementally numbered in the order that they're opened. Each tab is associated with its own Windows PowerShell console window. You can have up to 32 PowerShell tabs with their own session open at a time (this is limited to 8 on Windows PowerShell ISE 2.0.)

Note that clicking the **New** or **Open** icons on the toolbar doesn't create a new tab with a separate session. Instead, those buttons open a new or existing script file on the currently active tab with a session. You can have multiple script files open with each tab and session. The script tabs for a session only appear below the session tabs when the associated session is active.

To make a PowerShell tab active, click the tab. To select from all PowerShell tabs that are open, on the **View** menu, click the PowerShell tab you want to use.

To create and use a new Remote PowerShell tab

On the **File** menu, click **New Remote PowerShell Tab** to establish a session on a remote computer. A dialog box appears and prompts you to enter details required to establish the remote connection. The remote tab functions just like a local PowerShell tab, but the commands and scripts are run on the remote computer.

To close a PowerShell Tab

To close a tab, you can use any of the following techniques:

- Click the tab that you want to close.
- On the **File** menu, click **Close PowerShell Tab**, or click the Close button (X) on an active tab to close the tab.

If you have unsaved files open in the PowerShell tab that you are closing, you are prompted to save or discard them. For more information about how to save a script, see [How to Save a Script](#).

See Also

- [Introducing the Windows PowerShell ISE](#)
- [How to Use the Console Pane in the Windows PowerShell ISE](#)

How to Debug Scripts in Windows PowerShell ISE

Article • 03/27/2025

This article describes how to debug scripts on a local computer by using the Windows PowerShell Integrated Scripting Environment (ISE) visual debugging features.

How to manage breakpoints

A breakpoint is a designated spot in a script where you would like operation to pause so that you can examine the current state of the variables and the environment in which your script is running. Once your script is paused by a breakpoint, you can run commands in the Console Pane to examine the state of your script. You can output variables or run other commands. You can even modify the value of any variables that are visible to the context of the currently running script. After you have examined what you want to see, you can resume operation of the script.

You can set three types of breakpoints in the Windows PowerShell debugging environment:

1. **Line breakpoint.** The script pauses when the designated line is reached during the operation of the script
2. **Variable breakpoint.** The script pauses whenever the designated variable's value changes.
3. **Command breakpoint.** The script pauses whenever the designated command is about to be run during the operation of the script. It can include parameters to further filter the breakpoint to only the operation you want. The command can also be a function you created.

Of these, in the Windows PowerShell ISE debugging environment, only line breakpoints can be set by using the menu or the keyboard shortcuts. The other two types of breakpoints can be set, but they are set from the Console Pane by using the [Set-PSBreakpoint](#) cmdlet. This section describes how you can perform debugging tasks in Windows PowerShell ISE by using the menus where available, and perform a wider range of commands from the Console Pane by using scripting.

To set a breakpoint

A breakpoint can be set in a script only after it has been saved. Right-click the line where you want to set a line breakpoint, and then click **Toggle Breakpoint**. Or, click the line where you want to set a line breakpoint, and press **F9** or, on the **Debug** menu, click **Toggle Breakpoint**.

The following script is an example of how you can set a variable breakpoint from the Console Pane by using the [Set-PSBreakpoint](#) cmdlet.

PowerShell

```
# This command sets a breakpoint on the Server variable in the Sample.ps1
# script.
Set-PSBreakpoint -Script sample.ps1 -Variable Server
```

List all breakpoints

Displays all breakpoints in the current Windows PowerShell session.

On the **Debug** menu, click **List Breakpoints**. The following script is an example of how you can list all breakpoints from the Console Pane by using the [Get-PSBreakpoint](#) cmdlet.

PowerShell

```
# This command lists all breakpoints in the current session.
Get-PSBreakpoint
```

Remove a breakpoint

Removing a breakpoint deletes it.

If you think you might want to use it again later, consider [Disable a Breakpoint](#) instead. Right-click the line where you want to remove a breakpoint, and then click **ToggleBreakpoint**. Or, click the line where you want to remove a breakpoint, and on the **Debug** menu, click **Toggle Breakpoint**. The following script is an example of how to remove a breakpoint with a specified ID from the Console Pane by using the [Remove-PSBreakpoint](#) cmdlet.

PowerShell

```
# This command deletes the breakpoint with breakpoint ID 2.
Remove-PSBreakpoint -Id 2
```

Remove All Breakpoints

To remove all breakpoints defined in the current session, on the **Debug** menu, click **Remove All Breakpoints**.

The following script is an example of how to remove all breakpoints from the Console Pane by using the [Remove-PSBreakpoint](#) cmdlet.

PowerShell

```
# This command deletes all of the breakpoints in the current session.  
Get-PSBreakpoint | Remove-PSBreakpoint
```

Disable a Breakpoint

Disabling a breakpoint doesn't remove it. It turns it off until it's enabled. To disable a specific line breakpoint, right-click the line where you want to disable a breakpoint, and then click **Disable Breakpoint**.

Or, click the line where you want to disable a breakpoint, and press **F9** or, on the **Debug** menu, click **Disable Breakpoint**. The following script is an example of how you can remove a breakpoint with a specified ID from the Console Pane using the [Disable-PSBreakpoint](#) cmdlet.

PowerShell

```
# This command disables the breakpoint with breakpoint ID 0.  
Disable-PSBreakpoint -Id 0
```

Disable All Breakpoints

Disabling a breakpoint doesn't remove it; it turns it off until it's enabled. To disable all breakpoints in the current session, on the **Debug** menu, click **Disable all Breakpoints**. The following script is an example of how you can disable all breakpoints from the Console Pane by using the [Disable-PSBreakpoint](#) cmdlet.

PowerShell

```
# This command disables all breakpoints in the current session.  
# You can abbreviate this command as: "gbp | dbp".  
Get-PSBreakpoint | Disable-PSBreakpoint
```

Enable a Breakpoint

To enable a specific breakpoint, right-click the line where you want to enable a breakpoint, and then click **Enable Breakpoint**. Or, click the line where you want to enable a breakpoint, and then press **F9** or, on the **Debug** menu, click **Enable Breakpoint**. The following script is an example of how you can enable specific breakpoints from the Console Pane by using the [Enable-PSBreakpoint](#) cmdlet.

PowerShell

```
# This command enables breakpoints with breakpoint IDs 0, 1, and 5.  
Enable-PSBreakpoint -Id 0, 1, 5
```

Enable All Breakpoints

To enable all breakpoints defined in the current session, on the **Debug** menu, click **Enable all Breakpoints**. The following script is an example of how you can enable all breakpoints from the Console Pane by using the [Enable-PSBreakpoint](#) cmdlet.

PowerShell

```
# This command enables all breakpoints in the current session.  
# You can abbreviate the command by using their aliases: "gbp | ebp".  
Get-PSBreakpoint | Enable-PSBreakpoint
```

How to manage a debugging session

Before you start debugging, you must set one or more breakpoints. You can't set a breakpoint unless the script that you want to debug is saved. For directions on how to set a breakpoint, see [How to manage breakpoints](#) or [Set-PSBreakpoint](#). After you start debugging, you can't edit a script until you stop debugging. A script that has one or more breakpoints set is automatically saved before it's run.

To start debugging

Press **F5** or, on the toolbar, click the **Run Script** icon, or on the **Debug** menu click **Run/Continue**. The script runs until it encounters the first breakpoint. It pauses operation there and highlights the line on which it paused.

To continue debugging

Press **F5** or, on the toolbar, click the **Run Script** icon, or on the **Debug** menu, click **Run/Continue** or, in the **Console Pane**, type **C** and then press **ENTER**. This causes the script to continue running to the next breakpoint or to the end of the script if no further breakpoints are encountered.

To view the call stack

The call stack displays the current run location in the script. If the script is running in a function that was called by a different function, then that's represented in the display by additional rows in the output. The bottom-most row displays the original script and the line in it in which a function was called. The next line up shows that function and the line in it in which another function might have been called. The top-most row shows the current context of the current line on which the breakpoint is set.

While paused, to see the current call stack, press **CTRL + SHIFT + D** or, on the **Debug** menu, click **Display Call Stack** or, in the **Console Pane**, type **K** and then press **ENTER**.

To stop debugging

Press **SHIFT + F5** or, on the **Debug** menu, click **Stop Debugger**, or, in the **Console Pane**, type **Q** and then press **ENTER**.

How to step over, step into, and step out while debugging

Stepping is the process of running one statement at a time. You can stop on a line of code, and examine the values of variables and the state of the system. The following table describes common debugging tasks such as stepping over, stepping into, and stepping out.

[+] Expand table

Debugging Task	Description	How to accomplish it in PowerShell ISE
Step Into	Executes the current statement and then stops at the next statement. If the current statement is a function or script call, then the debugger steps into that function or script, otherwise it stops at the next statement.	Press F11 or, on the Debug menu, click Step Into , or in the Console Pane , type S and press ENTER .

Debugging Task	Description	How to accomplish it in PowerShell ISE
Step Over	Executes the current statement and then stops at the next statement. If the current statement is a function or script call, then the debugger executes the whole function or script, and it stops at the next statement after the function call.	Press F10 or, on the Debug menu, click Step Over , or in the Console Pane , type V and press ENTER .
Step Out	Steps out of the current function and up one level if the function is nested. If in the main body, the script is executed to the end, or to the next breakpoint. The skipped statements are executed, but not stepped through.	Press SHIFT + F11 , or on the Debug menu, click Step Out , or in the Console Pane , type O and press ENTER .
Continue	Continues execution to the end, or to the next breakpoint. The skipped functions and invocations are executed, but not stepped through.	Press F5 or, on the Debug menu, click Run/Continue , or in the Console Pane , type C and press ENTER .

How to display the values of variables while debugging

You can display the current values of variables in the script as you step through the code.

To display the values of standard variables

Use one of the following methods:

- In the Script Pane, hover over the variable to display its value as a tool tip.
- In the Console Pane, type the variable name and press **ENTER**.

All panes in ISE are always in the same scope. Therefore, while you are debugging a script, the commands that you type in the Console Pane run in script scope. This allows you to use the Console Pane to find the values of variables and call functions that are defined only in the script.

To display the values of automatic variables

You can use the preceding method to display the value of almost all variables while you are debugging a script. However, these methods don't work for the following automatic

variables.

- `$_`
- `$input`
- `$MyInvocation`
- `$PSBoundParameters`
- `$args`

If you try to display the value of any of these variables, you get the value of that variable for in an internal pipeline the debugger uses, not the value of the variable in the script. You can work around this for a few variables (`$_`, `$input`, `$MyInvocation`, `$PSBoundParameters`, and `$args`) by using the following method:

1. In the script, assign the value of the automatic variable to a new variable.
2. Display the value of the new variable, either by hovering over the new variable in the Script Pane, or by typing the new variable in the Console Pane.

For example, to display the value of the `$MyInvocation` variable, in the script, assign the value to a new variable, such as `$scriptName`, and then hover over or type the `$scriptName` variable to display its value.

PowerShell

```
# In C:\ps-test\MyScript.ps1
$scriptName = $MyInvocation.PSCommandPath
```

PowerShell

```
# In the Console Pane:
.\MyScript.ps1
$scriptName
```

Output

```
C:\ps-test\MyScript.ps1
```

See Also

[Exploring the Windows PowerShell ISE](#)

How to Use Profiles in Windows PowerShell ISE

Article • 03/27/2025

This article explains how to use Profiles in Windows PowerShell® Integrated Scripting Environment (ISE). We recommend that before performing the tasks in this section, you review [about_Profiles](#), or in the Console Pane, type, `Get-Help about_Profiles` and press `ENTER`.

A profile is a Windows PowerShell ISE script that runs automatically when you start a new session. You can create one or more Windows PowerShell profiles for Windows PowerShell ISE and use them to add the configure the Windows PowerShell or Windows PowerShell ISE environment, preparing it for your use, with variables, aliases, functions, and color and font preferences that you want available. A profile affects every Windows PowerShell ISE session that you start.

ⓘ Note

The Windows PowerShell execution policy determines whether you can run scripts and load a profile. The default execution policy, "Restricted," prevents all scripts from running, including profiles. If you use the "Restricted" policy, the profile can't load. For more information about execution policy, see [about_Execution_Policies](#).

Selecting a profile to use in the Windows PowerShell ISE

Windows PowerShell ISE supports profiles for the current user and all users. It also supports the Windows PowerShell profiles that apply to all hosts.

The profile that you use is determined by how you use Windows PowerShell and Windows PowerShell ISE.

- If you use only Windows PowerShell ISE to run Windows PowerShell, then save all your items in one of the ISE-specific profiles, such as the **CurrentUserCurrentHost** profile for Windows PowerShell ISE or the **AllUsersCurrentHost** profile for Windows PowerShell ISE.
- If you use multiple host programs to run Windows PowerShell, save your functions, aliases, variables, and commands in a profile that affects all host programs, such as

the `CurrentUserAllHosts` or the `AllUsersAllHosts` profile, and save ISE-specific features, like color and font customization in the `CurrentUserCurrentHost` profile for Windows PowerShell ISE profile or the `AllUsersCurrentHost` profile for Windows PowerShell ISE.

The following are profiles that can be created and used in Windows PowerShell ISE. Each profile is saved to its own specific path.

[+] Expand table

Profile Type	Profile Path
Current user, PowerShell ISE	<code>\$PROFILE.CurrentUserCurrentHost</code> , Or <code>\$PROFILE</code>
All users, PowerShell ISE	<code>\$PROFILE.AllUsersCurrentHost</code>
Current user, All hosts	<code>\$PROFILE.CurrentUserAllHosts</code>
All users, All hosts	<code>\$PROFILE.AllUsersAllHosts</code>

To create a new profile

To create a new "Current user, Windows PowerShell ISE" profile, run this command:

```
PowerShell

if (!(Test-Path -Path $PROFILE)) {
    New-Item -Type File -Path $PROFILE -Force
}
```

To create a new "All users, Windows PowerShell ISE" profile, run this command:

```
PowerShell

if (!(Test-Path -Path $PROFILE.AllUsersCurrentHost)) {
    New-Item -Type File -Path $PROFILE.AllUsersCurrentHost -Force
}
```

To create a new "Current user, All Hosts" profile, run this command:

```
PowerShell

if (!(Test-Path -Path $PROFILE.CurrentUserAllHosts)) {
    New-Item -Type File -Path $PROFILE.CurrentUserAllHosts -Force
}
```

To create a new "All users, All Hosts" profile, type:

PowerShell

```
if (!(Test-Path -Path $PROFILE.AllUsersAllHosts)) {  
    New-Item -Type File -Path $PROFILE.AllUsersAllHosts -Force  
}
```

To edit a profile

1. To open the profile, run the command `psEdit` with the variable that specifies the profile you want to edit. For example, to open the "Current user, Windows PowerShell ISE" profile, type: `psEdit $PROFILE`
2. Add some items to your profile. The following are a few examples to get you started:
 - To change the default background color of the Console Pane to blue, in the profile file type: `$psISE.Options.OutputPaneBackground = 'blue'`. For more information about the `$psISE` variable, see [Windows PowerShell ISE Object Model Reference](#).
 - To change font size to 20, in the profile file type: `$psISE.Options.FontSize =20`
3. To save your profile file, on the **File** menu, click **Save**. Next time you open the Windows PowerShell ISE, your customizations are applied.

See Also

- [about_Profiles](#)
- [Introducing the Windows PowerShell ISE](#)

How to Use Tab Completion in the Script Pane and Console Pane

Article • 03/27/2025

Tab completion provides automatic help when you are typing in the Script Pane or in the Command Pane. Use the following steps to take advantage of this feature:

To automatically complete a command entry

In the Command Pane or Script Pane, type a few characters of a command and then press `TAB` to select the desired completion text. If multiple items begin with the text that you initially typed, then continue pressing `TAB` until the item you want appears. Tab completion can help with typing a cmdlet name, parameter name, variable name, object property name, or a file path.

ⓘ Note

In the Script Pane, pressing `TAB` will automatically complete a command only when you are editing `.ps1`, `.psd1`, or `.psm1` files. Tab completion works any time when you are typing in the Command Pane.

To automatically complete a cmdlet parameter entry

In the Command Pane or Script pane, type a cmdlet followed by a dash and then press `TAB`.

For example, type `Get-Process -` and then press `TAB` multiple times to display each of the parameters for the cmdlet in turn.

See Also

- [Introducing the Windows PowerShell ISE](#)
- [How to Create a PowerShell Tab](#)

How to Use the Console Pane in the Windows PowerShell ISE

Article • 03/27/2025

The Console pane in the Windows PowerShell Integrated Scripting Environment (ISE) operates exactly like the stand-alone Windows PowerShell ISE console window.

To run a command in the Console Pane, type a command, and then press `ENTER`. To enter multiple commands that you want to execute in sequence, type `SHIFT + ENTER` between commands. See [How to Use Tab Completion in the Script Pane and Console Pane](#) for help in typing commands.

To stop a command, on the toolbar, click **Stop Operation**, or press `CTRL + BREAK`. You can also use `CTRL + C` to stop a command if the context is unambiguous. For example, if some text has been selected in the current Pane, then `CTRL + C` maps to the copy operation.

Beginning in Windows PowerShell v3, the Output pane was combined with the Console pane. This has the benefit of behaving like the stand-alone Windows PowerShell console and eliminates the differences in procedures that were needed when they were separate. You can:

- Select and copy text from the Console pane to the Clipboard for pasting in any other window. To select text, click and hold the mouse in the output pane while dragging the mouse over the text you want to capture. You can also use the cursor arrow keys while holding `SHIFT` to select text. Then press `CTRL + C` or click the **Copy** icon in the toolbar.
- Paste the selected text at a current cursor position. Click the **Paste** icon on the toolbar.
- Clear all the text in the Console pane. To clear the Console pane, you can click the **Clear Console Pane** icon on the toolbar, or run the command `Clear-Host` or its alias, `cls`.

See Also

- [Introducing the Windows PowerShell ISE](#)

How to Write and Run Scripts in the Windows PowerShell ISE

Article • 03/27/2025

This article describes how to create, edit, run, and save scripts in the Script Pane.

How to create and run scripts

You can open and edit Windows PowerShell files in the Script Pane. Specific file types of interest in Windows PowerShell are script files (`.ps1`), script data files (`.psd1`), and script module files (`.psm1`). These file types are syntax colored in the Script Pane editor. Other common file types you may open in the Script Pane are configuration files (`.ps1xml`), XML files, and text files.

ⓘ Note

The Windows PowerShell execution policy determines whether you can run scripts and load Windows PowerShell profiles and configuration files. The default execution policy, Restricted, prevents all scripts from running, and prevents loading profiles. To change the execution policy to allow profiles to load and be used, see [Set-ExecutionPolicy](#) and [about_Signing](#).

To create a new script file

On the toolbar, click **New**, or on the **File** menu, click **New**. The created file appears in a new file tab under the current PowerShell tab. Remember that the PowerShell tabs are only visible when there are more than one. By default a file of type script (`.ps1`) is created, but it can be saved with a new name and extension. Multiple script files can be created in the same PowerShell tab.

To open an existing script

On the toolbar, click **Open**, or on the **File** menu, click **Open**. In the **Open** dialog box, select the file you want to open. The opened file appears in a new tab.

To close a script tab

Click the **Close** icon (X) of the file tab you want to close or select the **File** menu and click **Close**.

If the file has been altered since it was last saved, you're prompted to save or discard it.

To display the file path

On the file tab, point to the file name. The fully qualified path to the script file appears in a tooltip.

To run a script

On the toolbar, click **Run Script**, or on the **File** menu, click **Run**.

To run a portion of a script

1. In the Script Pane, select a portion of a script.
2. On the **File** menu, click **Run Selection**, or on the toolbar, click **Run Selection**.

To stop a running script

There are several ways to stop a running script.

- Click **Stop Operation** on the toolbar
- Press **CTRL + BREAK**
- Select the **File** menu and click **Stop Operation**.

Pressing **CTRL + C** also works unless some text is currently selected, in which case **CTRL + C** maps to the copy function for the selected text.

How to write and edit text in the Script Pane

You can copy, cut, paste, find, and replace text in the Script Pane. You can also undo and redo the last action you just performed. The keyboard shortcuts for these actions are the same shortcuts used for all Windows applications.

To enter text in the Script Pane

1. Move the cursor to the Script Pane by clicking anywhere in the Script Pane, or by clicking **Go to Script Pane** in the **View** menu.

2. Create a script. Syntax coloring and tab completion provide a richer editing experience in Windows PowerShell ISE.
3. See [How to Use Tab Completion in the Script Pane and Console Pane](#) for details about using the tab completion feature to help in typing.

To find text in the Script Pane

1. To find text anywhere, press **CTRL + F** or, on the **Edit** menu, click **Find in Script**.
2. To find text after the cursor, press **F3** or, on the **Edit** menu, click **Find Next in Script**.
3. To find text before the cursor, press **SHIFT + F3** or, on the **Edit** menu, click **Find Previous in Script**.

To find and replace text in the Script Pane

Press **CTRL + H** or, on the **Edit** menu, click **Replace in Script**. Enter the text you want to find and the replacement text, then press **ENTER**.

To go to a particular line of text in the Script Pane

1. In the Script Pane, press **CTRL + G** or, on the **Edit** menu, click **Go to Line**.
2. Enter a line number.

To copy text in the Script Pane

1. In the Script Pane, select the text that you want to copy.
2. Press **CTRL + C** or, on the toolbar, click the **Copy** icon, or on the **Edit** menu, click **Copy**.

To cut text in the Script Pane

1. In the Script Pane, select the text that you want to cut.
2. Press **CTRL + X** or, on the toolbar, click the **Cut** icon, or on the **Edit** menu, click **Cut**.

To paste text into the Script Pane

Press **CTRL + V** or, on the toolbar, click the **Paste** icon, or on the **Edit** menu, click **Paste**.

To undo an action in the Script Pane

Press **CTRL + Z** or, on the toolbar, click the **Undo** icon, or on the **Edit** menu, click **Undo**.

To redo an action in the Script Pane

Press **CTRL + Y** or, on the toolbar, click the **Redo** icon, or on the **Edit** menu, click **Redo**.

How to save a script

An asterisk appears next to the script name to mark a file that hasn't been saved since it was changed. The asterisk disappears when the file is saved.

To save a script

Press **CTRL + S** or, on the toolbar, click the **Save** icon, or on the **File** menu, click **Save**.

To save and name a script

1. On the **File** menu, click **Save As**. The **Save As** dialog box will appear.
2. In the **File name** box, enter a name for the file.
3. In the **Save as type** box, select a file type. For example, in the **Save as type** box, select '**PowerShell Scripts (*.ps1)**'.
4. Click **Save**.

To save a script in ASCII encoding

By default, Windows PowerShell ISE saves new script files (**.ps1**), script data files (**.psd1**), and script module files (**.psm1**) as Unicode (BigEndianUnicode). To save a script in another encoding, such as ASCII (ANSI), use the **Save** or **SaveAs** methods on the **\$psISE.CurrentFile** object.

The following command saves a new script as **MyScript.ps1** with ASCII encoding.

```
PowerShell
```

```
$psISE.CurrentFile.SaveAs("MyScript.ps1", [System.Text.Encoding]::ASCII)
```

The following command replaces the current script file with a file with the same name, but with ASCII encoding.

```
PowerShell
```

```
$psISE.CurrentFile.Save([System.Text.Encoding]::ASCII)
```

The following command gets the encoding of the current file.

```
PowerShell
```

```
$psISE.CurrentFile.encoding
```

Windows PowerShell ISE supports the following encoding options: ASCII, BigEndianUnicode, Unicode, UTF32, UTF7, UTF8, and Default. The value of the Default option varies with the system.

Windows PowerShell ISE doesn't change the encoding of script files when you use the Save or Save As commands.

See Also

- [Exploring the Windows PowerShell ISE](#)

Keyboard Shortcuts for the Windows PowerShell ISE

Article • 03/27/2025

Use the following keyboard shortcuts to perform actions in Windows PowerShell Integrated Scripting Environment (ISE). Windows PowerShell ISE is available as part of the Windows Server and Windows client operating systems.

Keyboard shortcuts for editing text

You can use the following keyboard shortcuts when you edit text.

[+] Expand table

Action	Keyboard Shortcuts	Use in
Help	F1	Script Pane Important: You can specify that F1 help comes from Microsoft Learn or downloaded Help (see Update-Help). To select, click Tools, Options, then on the General Settings tab, set or clear Use local help content instead of online content .
Select All	CTRL + A	Script Pane
Copy	CTRL + C	Script Pane, Command Pane, Output Pane
Cut	CTRL + X	Script Pane, Command Pane
Expand or Collapse Outlining	CTRL + M	Script Pane
Find in Script	CTRL + F	Script Pane
Find Next in Script	F3	Script Pane
Find Previous in Script	SHIFT + F3	Script Pane
Find Matching Brace	CTRL +]	Script Pane
Paste	CTRL + V	Script Pane, Command Pane

Action	Keyboard Shortcuts	Use in
Make Lowercase	<code>CTRL + U</code>	Script Pane, Command Pane
Make Uppercase	<code>CTRL + SHIFT + U</code>	Script Pane, Command Pane
Redo	<code>CTRL + Y</code>	Script Pane, Command Pane
Replace in Script	<code>CTRL + H</code>	Script Pane
Save	<code>CTRL + S</code>	Script Pane
Select All	<code>CTRL + A</code>	Script Pane, Command Pane, Output Pane
Show Snippets	<code>CTRL + J</code>	Script Pane, Command Pane
Undo	<code>CTRL + Z</code>	Script Pane, Command Pane
Show Intellisense Help	<code>CTRL + Space</code>	Script Pane
Delete word to left	<code>CTRL + Backspace</code>	Script Pane
Delete word to right	<code>CTRL + Delete</code>	Script Pane

Keyboard shortcuts for running scripts

You can use the following keyboard shortcuts when you run scripts in the Script Pane.

[\[+\] Expand table](#)

Action	Keyboard Shortcut
New	<code>CTRL + N</code>
Open	<code>CTRL + O</code>
Run	<code>F5</code>
Run Selection	<code>F8</code>

Action	Keyboard Shortcut
Stop Execution	<code>CTRL + BREAK</code> . <code>CTRL + C</code> can be used when the context is unambiguous (when there is no text selected).
Tab (to next script)	<code>CTRL + TAB</code> Note: Tab to next script works only when you have a single Windows PowerShell tab open, or when you have more than one Windows PowerShell tab open, but the focus is in the Script Pane.
Tab (to previous script)	<code>CTRL + SHIFT + TAB</code> Note: Tab to previous script works when you have only one Windows PowerShell tab open, or if you have more than one Windows PowerShell tab open, and the focus is in the Script Pane.

Keyboard shortcuts for customizing the view

You can use the following keyboard shortcuts to customize the view in Windows PowerShell ISE. They are accessible from all the panes in the application.

[Expand table](#)

Action	Keyboard Shortcut
Go to Command (v2) or Console (v3 and later) Pane	<code>CTRL + D</code>
Go to Output Pane (v2 only)	<code>CTRL + SHIFT + O</code>
Go to Script Pane	<code>CTRL + I</code>
Show Script Pane	<code>CTRL + R</code>
Hide Script Pane	<code>CTRL + R</code>
Move Script Pane Up	<code>CTRL + 1</code>
Move Script Pane Right	<code>CTRL + 2</code>
Maximize Script Pane	<code>CTRL + 3</code>
Zoom In	<code>CTRL + +</code>
Zoom Out	<code>CTRL + -</code>

Keyboard shortcuts for debugging scripts

You can use the following keyboard shortcuts when you debug scripts.

[Expand table](#)

Action	Keyboard Shortcut	Use in
Run/Continue	F5	Script Pane, when debugging a script
Step Into	F11	Script Pane, when debugging a script
Step Over	F10	Script Pane, when debugging a script
Step Out	SHIFT + F11	Script Pane, when debugging a script
Display Call Stack	CTRL + SHIFT + D	Script Pane, when debugging a script
List Breakpoints	CTRL + SHIFT + L	Script Pane, when debugging a script
Toggle Breakpoint	F9	Script Pane, when debugging a script
Remove All Breakpoints	CTRL + SHIFT + F9	Script Pane, when debugging a script
Stop Debugger	SHIFT + F5	Script Pane, when debugging a script

Note

You can also use the keyboard shortcuts designed for the Windows PowerShell console when you debug scripts in Windows PowerShell ISE. To use these shortcuts, you must type the shortcut in the Command Pane and press `ENTER`.

[Expand table](#)

Action	Keyboard Shortcut	Use in
Continue	C	Console Pane, when debugging a script
Step Into	S	Console Pane, when debugging a script
Step Over	V	Console Pane, when debugging a script
Step Out	O	Console Pane, when debugging a script
Repeat Last Command (for Step Into or Step Over)	ENTER	Console Pane, when debugging a script

Action	Keyboard Shortcut	Use in
Display Call Stack	K	Console Pane, when debugging a script
Stop Debugging	Q	Console Pane, when debugging a script
List the Script	L	Console Pane, when debugging a script
Display Console Debugging Commands	H or ?	Console Pane, when debugging a script

Keyboard shortcuts for Windows PowerShell tabs

You can use the following keyboard shortcuts when you use Windows PowerShell tabs.

[Expand table](#)

Action	Keyboard Shortcut
Close PowerShell Tab	CTRL + W
New PowerShell Tab	CTRL + T
Previous PowerShell tab	CTRL + SHIFT + TAB . This shortcut works only when no files are open on any Windows PowerShell tab.
Next Windows PowerShell tab	CTRL + TAB . This shortcut works only when no files are open on any Windows PowerShell tab.

Keyboard shortcuts for starting and exiting

You can use the following keyboard shortcuts to exit the Windows PowerShell ISE or to start a new Windows PowerShell session outside of the ISE.

[Expand table](#)

Action	Keyboard Shortcut
Exit	ALT + F4 closes the ISE.

Action	Keyboard Shortcut
Start powershell.exe	<code>CTRL + SHIFT + P</code> opens a new Windows PowerShell session outside of the ISE.

See Also

- [PowerShell Magazine: The Complete List of Windows PowerShell ISE Keyboard Shortcuts ↗](#)

Accessibility in Windows PowerShell ISE

Article • 03/27/2025

This topic describes the accessibility features of Windows PowerShell Integrated Scripting Environment (ISE) that you might find helpful.

- [How to change the size and location of the Console and Script Panes](#)
- [Keyboard shortcuts for editing text](#)
- [Keyboard shortcuts for running scripts](#)
- [Keyboard shortcuts for customizing the view](#)
- [Keyboard shortcuts for debugging scripts](#)
- [Keyboard shortcuts for Windows PowerShell tabs](#)
- [Keyboard shortcuts for starting and exiting](#)
- [Breakpoint management with cmdlets](#)

Microsoft is committed to making its products and services easier for everyone to use. The following topics provide information about the features, products, and services that make Windows PowerShell ISE more accessible for people with disabilities.

In addition to accessibility features and utilities in Microsoft Windows, the following features make Windows PowerShell ISE more accessible for people with disabilities:

- Keyboard Shortcuts
- Syntax Coloring Table and the ability to modify several other color settings using the `$psISE.Options` scripting object.
- Text Size Change

How to change the size and location of the Console and Script Panes

You can use the following steps to change the size and location of the Console Pane and the Script Pane. When you open the Windows PowerShell ISE again, the size and location changes you made will be retained.

To resize the Script Pane and Console Pane

1. Pause the pointer on the split line between the Script Pane and Console Pane.

- When the mouse pointer changes to a two-headed arrow, drag the border to change the size of the pane.

To move the Script Pane and Console Pane

Do one of the following:

- To move the Script Pane above the Console Pane, press **CTRL + 1** or, on the toolbar, click the **Show Script Pane Top** icon, or in the **View** menu, click **Show Script Pane Top**.
- To move the Script Pane to the right of the Console Pane, press **CTRL + 2** or, on the toolbar, click the **Show Script Pane Right** icon, or in the **View** menu, click **Show Script Pane Right**.
- To maximize the Script Pane, press **CTRL + 3** or, on the toolbar, click the **Show Script Pane Maximized** icon, or in the **View** menu, click **Show Script Pane Maximized**.
- To maximize the Console Pane and hide the Script Pane, on the far right edge of the row of tabs, click the **Hide Script Pane** icon, in the **View** menu, click to deselect the **Show Script Pane** menu option.
- To display the Script Pane when the Console Pane is maximized, on the far right edge of the row of tabs, click the **Show Script Pane** icon, or in the **View** menu, click to select the **Show Script Pane** menu option.

Keyboard shortcuts for editing text

You can use the following keyboard shortcuts when you edit text.

 Expand table

Action	Keyboard Shortcuts	Use in
Copy	CTRL + C	Script Pane, Console Pane
Cut	CTRL + X	Script Pane, Console Pane
Find in Script	CTRL + F	Script Pane
Find Next in Script	F3	Script Pane
Find Previous in Script	SHIFT + F3	Script Pane

Action	Keyboard Shortcuts	Use in
Paste	<code>CTRL + V</code>	Script Pane, Console Pane
Redo	<code>CTRL + Y</code>	Script Pane, Console Pane
Replace in Script	<code>CTRL + H</code>	Script Pane
Save	<code>CTRL + S</code>	Script Pane
Select All	<code>CTRL + A</code>	Script Pane, Console Pane
Undo	<code>CTRL + Z</code>	Script Pane, Console Pane

Keyboard shortcuts for running scripts

You can use the following keyboard shortcuts when you run scripts in the Script Pane.

[\[+\] Expand table](#)

Action	Keyboard Shortcut
New	<code>CTRL + N</code>
Open	<code>CTRL + O</code>
Run	<code>F5</code>
Run Selection	<code>F8</code>
Stop Execution	<code>CTRL + BREAK</code> . <code>CTRL + C</code> can be used when the context is unambiguous (when there is no text selected).
Tab (to next script)	<code>CTRL + TAB</code> Note: Tab to next script works only when you have a single PowerShell tab open, or when you have more than one PowerShell tab open, but the focus is in the Script Pane.
Tab (to previous script)	<code>CTRL + SHIFT + TAB</code> Note: Tab to previous script works when you have only one PowerShell tab open, or if you have more than one PowerShell tab open, and the focus is in the Script Pane.

Keyboard shortcuts for customizing the view

You can use the following keyboard shortcuts to customize the view in Windows PowerShell ISE. They are accessible from all the panes in the application.

[\[+\] Expand table](#)

Action	Keyboard Shortcut
Go to Console Pane	CTRL + D
Go to Script Pane	CTRL + I
Show Script Pane	CTRL + R
Hide Script Pane	CTRL + R
Move Script Pane Up	CTRL + 1
Move Script Pane Right	CTRL + 2
Maximize Script Pane	CTRL + 3
Zoom In	CTRL + PLUS
Zoom Out	CTRL + MINUS

Keyboard shortcuts for debugging scripts

You can use the following keyboard shortcuts when you debug scripts.

[\[+\] Expand table](#)

Action	Keyboard Shortcut	Use in
Run/Continue	F5	Script Pane, when debugging a script
Step Into	F11	Script Pane, when debugging a script
Step Over	F10	Script Pane, when debugging a script
Step Out	SHIFT + F11	Script Pane, when debugging a script
Display Call Stack	CTRL + SHIFT + D	Script Pane, when debugging a script
List Breakpoints	CTRL + SHIFT + L	Script Pane, when debugging a script
Toggle Breakpoint	F9	Script Pane, when debugging a script
Remove All Breakpoints	CTRL + SHIFT + F9	Script Pane, when debugging a script
Stop Debugger	SHIFT + F5	Script Pane, when debugging a script

 **Note**

You can also use the keyboard shortcuts designed for the Windows PowerShell console when you debug scripts in Windows PowerShell ISE. To use these shortcuts, you must type the shortcut in the Console Pane and press **ENTER**.

 [Expand table](#)

Action	Keyboard Shortcut	Use in
Continue	C	Console Pane, when debugging a script
Step Into	S	Console Pane, when debugging a script
Step Over	V	Console Pane, when debugging a script
Step Out	O	Console Pane, when debugging a script
Repeat Last Command(Step Into/Over)	ENTER	Console Pane, when debugging a script
Display Call Stack	K	Console Pane, when debugging a script
Stop Debugging	Q	Console Pane, when debugging a script
List the Script	L	Console Pane, when debugging a script
Display Console Debugging Commands	H or ?	Console Pane, when debugging a script

Keyboard shortcuts for Windows PowerShell tabs

You can use the following keyboard shortcuts when you use Windows PowerShell tabs.

 [Expand table](#)

Action	Keyboard Shortcut
Close PowerShell Tab	<code>CTRL + W</code>
New PowerShell Tab	<code>CTRL + T</code>
Previous PowerShell tab	<code>CTRL + SHIFT + TAB</code> (Only when no files are open on any PowerShell tab)
Next Windows PowerShell tab	<code>CTRL + TAB</code> (Only when no files are open on any PowerShell tab)

Keyboard shortcuts for starting and exiting

You can use the following keyboard shortcuts to start the Windows PowerShell console (`powershell.exe`) or to exit Windows PowerShell ISE.

[\[+\] Expand table](#)

Action	Keyboard Shortcut
Exit	<code>ALT + F4</code>
Start <code>powershell.exe</code> (Windows PowerShell console)	<code>CTRL + SHIFT + P</code>

Breakpoint Management

For the visually impaired, breakpoint information is available through the cmdlets for managing breakpoints, such as [Get-PSBreakpoint](#) and [Set-PSBreakpoint](#). For more information please see 'How to manage breakpoints' in [How to Debug Scripts in the Windows PowerShell ISE](#).

See Also

[Introducing the Windows PowerShell ISE](#)

Purpose of the Windows PowerShell ISE Scripting Object Model

Article • 03/27/2025

Objects are associated with the form and function of Windows PowerShell Integrated Scripting Environment (ISE). The object model reference provides details about the member properties and methods that these objects expose. Examples are provided to show how you can use scripts to directly access these methods and properties. The scripting object model makes the following range of tasks easier.

Customizing the appearance of Windows PowerShell ISE

You can use the object model to modify the application settings and options. For example, you can modify them as follows:

- Change the color of errors, warnings, verbose outputs, and debug outputs.
- Get or set the background colors for the Command pane, the Output pane, and the Script pane.
- Set the foreground color for the Output pane.
- Set the font name and font size for Windows PowerShell ISE.
- Configure warnings. This setting includes warnings that are issued when a file is opened in multiple PowerShell tabs or when a script in the file is run before the file has been saved.
- Switch between a view where the Script pane and the Output pane are side-by-side and a view where the Script pane is on top of the Output pane.
- Dock the Command pane to the bottom or the top of the Output pane.

Enhancing the functionality of Windows PowerShell ISE

You can use the object model to enhance the functionality of Windows PowerShell ISE. For example, you can:

- Add and modify the instance of Windows PowerShell ISE itself. For example, to change the menus, you can add new menu items and map the new menu items to scripts.

- Create scripts that perform some of the tasks that you can perform by using the menu commands and buttons in Windows PowerShell ISE. For example, you can add, remove, or select a PowerShell tab.
- Complement tasks that can be performed by using menu commands and buttons. For example, you can rename a PowerShell tab.
- Manipulate text buffers for the Command pane, the Output pane, and the Script pane that are associated with a file. For example, you can:
 - Get or set all text.
 - Get or set a text selection.
 - Run a script or run a selected portion of a script.
 - Scroll a line into view.
 - Insert text at a caret position.
 - Select a block of text.
 - Get the last line number.
- Perform file operations. For example, you can:
 - Open a file, save a file, or save a file by using a different name.
 - Determine whether a file has been changed after it was last saved.
 - Get the file name.
 - Select a file.

Automating tasks

You can use the scripting object model to create keyboard shortcuts for frequent operations.

See also

- [The ISE Object Model Hierarchy](#)

The ISE Object Model Hierarchy

Article • 03/27/2025

This article shows the hierarchy of objects that are part of Windows PowerShell Integrated Scripting Environment (ISE). Windows PowerShell ISE is included in Windows PowerShell 3.0, 4.0, and 5.1. Click an object to take you to the reference documentation for the class that defines the object.

\$psISE Object

The `$psISE` object is the [root object](#) of the Windows PowerShell ISE object hierarchy. Located at the top level, it makes the following objects available for scripting:

\$psISE.CurrentFile

The `$psISE.CurrentFile` object is an instance of the [ISEFile](#) class.

\$psISE.CurrentPowerShellTab

The `$psISE.CurrentPowerShellTab` object is an instance of the [PowerShellTab](#) class.

\$psISE.CurrentVisibleHorizontalTool

The `$psISE.CurrentVisibleHorizontalTool` object is an instance of the [ISEAddOnTool](#) class. It represents the installed add-on tool that's currently docked to the top edge of the Windows PowerShell ISE window.

\$psISE.CurrentVisibleVerticalTool

The `$psISE.CurrentVisibleVerticalTool` object is an instance of the [ISEAddOnTool](#) class. It represents the installed add-on tool that's currently docked to the right-hand edge of the Windows PowerShell ISE window.

\$psISE.Options

The `$psISE.Options` object is an instance of the [ISEOptions](#) class. The ISEOptions object represents various settings for Windows PowerShell ISE. It's an instance of the

Microsoft.PowerShell.Host.ISE.ISEOptions class.

\$psISE.PowerShellTabs

The `$psISE.PowerShellTabs` object is an instance of the [PowerShellTabCollection](#) class. It's a collection of all the currently open PowerShell tabs that represent the available Windows PowerShell run environments on the local computer or on connected remote computers. Each member in the collection is an instance of the [PowerShellTab](#) class.

See Also

- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The ObjectModelRoot Object

Article • 03/27/2025

The `$psISE` object, which is the principal root object in Windows PowerShell Integrated Scripting Environment (ISE) is an instance of the `Microsoft.PowerShell.Host.ISE.ObjectModelRoot` class. This topic describes the properties of the `.ObjectModelRoot` object.

Properties

CurrentFile

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the file, which is associated with this host object that currently has the focus.

CurrentPowerShellTab

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the PowerShell tab that has the focus.

CurrentVisibleHorizontalTool

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the currently visible Windows PowerShell ISE add-on tool that's located in the horizontal tool pane at the bottom of the editor.

CurrentVisibleVerticalTool

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the currently visible Windows PowerShell ISE add-on tool that's located in the vertical tool pane on the right side of the editor.

Options

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the various options that can change settings in Windows PowerShell ISE.

PowerShellTabs

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the collection of the PowerShell tabs, which are open in Windows PowerShell ISE. By default, this object contains one PowerShell tab. However, you can add more PowerShell tabs to this object by using scripts or by using the menus in Windows PowerShell ISE.

See Also

- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The ISEAddOnToolCollection Object

Article • 03/27/2025

The **ISEAddOnToolCollection** object is a collection of **ISEAddOnTool** objects. An example is the `$psISE.CurrentPowerShellTab.VerticalAddOnTools` object.

Methods

Add(Name, ControlType, [IsVisible])

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Adds a new add-on tool to the collection. It returns the newly added add-on tool.

Before you run this command, you must install the add-on tool on the local computer and load the assembly.

- **Name** - String - Specifies the display name of the add-on tool that's added to Windows PowerShell ISE.
- **ControlType** - Type - Specifies the control that's added.
- **[IsVisible]** - optional Boolean - If set to `$true`, the add-on tool is immediately visible in the associated tool pane.

PowerShell

```
# Load a DLL with an add-on and then add it to the ISE
[Reflection.Assembly]::LoadFile("C:\testISESimpleSolution\ISESimpleSolution.dll")
$psISE.CurrentPowerShellTab.VerticalAddOnTools.Add("Solutions",
[ISESimpleSolution.Solution], $true)
```

Remove(Item)

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Removes the specified add-on tool from the collection.

- **Item** - Microsoft.PowerShell.Host.ISE.ISEAddOnTool - Specifies the object to be removed from Windows PowerShell ISE.

PowerShell

```
# Load a DLL with an add-on and then add it to the ISE
[Reflection.Assembly]::LoadFile("C:\test\ISESimpleSolution\ISESimpleSolution
.dll")
$psISE.CurrentPowerShellTab.VerticalAddOnTools.Add("Solutions",
[ISESimpleSolution.Solution], $true)
```

SetSelectedPowerShellTab(psTab)

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Selects the PowerShell tab that the **psTab** parameter specifies.

- **psTab** - Microsoft.PowerShell.Host.ISE.PowerShellTab -The PowerShell tab to select.

PowerShell

```
$newTab = $psISE.PowerShellTabs.Add()
# Change the DisplayName of the new PowerShell tab.
$newTab.DisplayName = 'Brand New Tab'
```

Remove(psTab)

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Removes the PowerShell tab that the **psTab** parameter specifies.

- **psTab** - Microsoft.PowerShell.Host.ISE.PowerShellTab - The PowerShell tab to remove.

PowerShell

```
$newTab = $psISE.PowerShellTabs.Add()
Change the DisplayName of the new PowerShell tab.
$newTab.DisplayName = 'This tab will go away in 5 seconds'
sleep 5
$psISE.PowerShellTabs.Remove($newTab)
```

See Also

- [The PowerShellTab Object](#)
- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The ISEAddOnTool Object

Article • 03/27/2025

An **ISEAddonTool** object represents an installed add-on tool that provides additional functionality to Windows PowerShell ISE. An example is the **Commands** tool that you can display by clicking **View**, then **Show Command Add-on**. This tool is then accessible to you by manipulating the various available **ISEAddOnTool** objects.

Each add-on tool can be associated with either the vertical pane or the horizontal pane. The vertical pane is docked to the right edge of Windows PowerShell ISE. The horizontal pane is docked to the bottom edge.

Each PowerShell tab in Windows PowerShell ISE can have its own set of add-on tools installed. See [\\$psISE.CurrentPowerShellTab.HorizontalAddOnTools](#) and [\\$psISE.CurrentPowerShellTab.VerticalAddOnTools](#) to access the collection of tools available to the currently selected tab or the same properties on any of the **PowerShellTab** objects in the [\\$psISE.PowerShellTabs](#) collection object.

Methods

There are no Windows PowerShell ISE-specific methods available for objects of this class.

Properties

Control

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

The **Control** property provides read access to many of the details of the Commands add-on tool.

PowerShell

```
# View the properties of the Commands add-on tool.  
# (assumes that it's visible in the vertical pane)  
$psISE.CurrentVisibleVerticalTool.Control
```

Output

```
HostObject           : Microsoft.PowerShell.Host.ISE.ObjectModelRoot
Content             :
HasContent          :
ContentTemplate     :
ContentTemplateSelector :
ContentStringFormat   :
BorderBrush          :
BorderThickness       :
Background            :
Foreground            :
FontFamily            :
FontSize              :
FontStretch           :
FontStyle              :
FontWeight            :
HorizontalContentAlignment :
VerticalContentAlignment :
TabIndex              :
IsTabStop             :
Padding               :
Template              : System.Windows.Controls.ControlTemplate
Style                :
OverridesDefaultCellStyle   :
UseLayoutRounding      :
Triggers              : {}
TemplatedParent        :
Resources             : {System.Windows.Controls.TabItem}
DataContext          :
BindingGroup          :
Language              :
Name                 :
Tag                  :
InputScope            :
ActualWidth           : 370.75
ActualHeight          : 676.559097412109
LayoutTransform        :
Width                 :
MinWidth              :
MaxWidth              :
Height                :
MinHeight              :
MaxHeight              :
FlowDirection          : LeftToRight
Margin                :
HorizontalAlignment       :
VerticalAlignment       :
FocusVisualStyle       :
Cursor                :
ForceCursor            :
IsInitialized          : True
IsLoaded              :
ToolTip                :
ContextMenu            :
Parent                :
```

```
HasAnimatedProperties      :  
InputBindings             :  
CommandBindings          :  
AllowDrop                 :  
DesiredSize               : 227.66,676.559097412109  
IsMeasureValid           : True  
IsArrangeValid           : True  
RenderSize                : 370.75,676.559097412109  
RenderTransform            :  
RenderTransformOrigin      :  
IsMouseDirectlyOver       : False  
IsMouseOver                : False  
IsStylusOver              : False  
IsKeyboardFocusWithin      : False  
IsMouseCaptured           :  
IsMouseCaptureWithin       : False  
IsStylusDirectlyOver      : False  
IsStylusCaptured          :  
IsStylusCaptureWithin      : False  
IsKeyboardFocused          : False  
IsInputMethodEnabled       :  
Opacity                   :  
OpacityMask                :  
BitmapEffect               :  
Effect                     :  
BitmapEffectInput          :  
CacheMode                  :  
Uid                        :  
Visibility                 : Visible  
ClipToBounds               : False  
Clip                      :  
SnapsToDevicePixels         : False  
IsFocused                  :  
IsEnabled                  :  
IsHitTestVisible           :  
IsVisible                  : True  
Focusable                  :  
PersistId                  : 1  
IsManipulationEnabled      :  
AreAnyTouchesOver           : False  
AreAnyTouchesDirectlyOver   :  
AreAnyTouchesCapturedWithin : False  
AreAnyTouchesCaptured       :  
TouchesCaptured             : {}  
TouchesCapturedWithin       : {}  
TouchesOver                 : {}  
TouchesDirectlyOver         : {}  
DependencyObjectType        : System.Windows.DependencyObjectType  
IsSealed                   : False  
Dispatcher                 : System.Windows.Threading.Dispatcher
```

isVisible

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

The Boolean property that indicates whether the add-on tool is currently visible in its assigned pane. If it's visible, you can set the **IsVisible** property to `$false` to hide the tool, or set the **IsVisible** property to `$true` to make an add-on tool visible on its PowerShell tab. Note that after an add-on tool is hidden, it's no longer accessible through the **CurrentVisibleHorizontalTool** or **CurrentVisibleVerticalTool** objects, and therefore can't be made visible by using this property on that object.

PowerShell

```
# Hide the current tool in the vertical tool pane
$psISE.CurrentVisibleVerticalTool.IsVisible = $false
# Show the first tool on the currently selected PowerShell tab
$psISE.CurrentPowerShellTab.VerticalAddOnTools[0].IsVisible = $true
```

Name

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

The read-only property that gets the name of the add-on tool.

PowerShell

```
# Gets the name of the visible vertical pane add-on tool.
$psISE.CurrentVisibleVerticalTool.Name
```

Output

Commands

See Also

- [The ISEAddOnToolCollection Object](#)
- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The ISEEditor Object

Article • 03/27/2025

An **ISEEditor** object is an instance of the `Microsoft.PowerShell.Host.ISE.ISEEditor` class. The Console pane is an **ISEEditor** object. Each **ISEFile** object has an associated **ISEEditor** object. The following sections list the methods and properties of an **ISEEditor** object.

Methods

Clear()

Supported in Windows PowerShell ISE 2.0 and later.

Clears the text in the editor.

PowerShell

```
# Clears the text in the Console pane.  
$psISE.CurrentPowerShellTab.ConsolePane.Clear()
```

EnsureVisible(int lineNumber)

Supported in Windows PowerShell ISE 2.0 and later.

Scrolls the editor so that the line that corresponds to the specified `lineNumber` parameter value is visible. It throws an exception if the specified line number is outside the range of 1,last line number, which defines the valid line numbers.

- `lineNumber` - The number of the line that's to be made visible.

PowerShell

```
# Scrolls the text in the Script pane so that the fifth line is in view.  
$psISE.CurrentFile.Editor.EnsureVisible(5)
```

Focus()

Supported in Windows PowerShell ISE 2.0 and later.

Sets the focus to the editor.

PowerShell

```
# Sets focus to the Console pane.  
$psISE.CurrentPowerShellTab.ConsolePane.Focus()
```

GetLineLength(int lineNumber)

Supported in Windows PowerShell ISE 2.0 and later.

Gets the line length as an integer for the line that's specified by the line number.

- **lineNumber** - The number of the line of which to get the length.
- **Returns** - The line length for the line at the specified line number.

PowerShell

```
# Gets the length of the first line in the text of the Command pane.  
$psISE.CurrentPowerShellTab.ConsolePane.GetLineLength(1)
```

GoToMatch()

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Moves the caret to the matching character if the **CanGoToMatch** property of the editor object is `$true`, which occurs when the caret is immediately before an opening parenthesis, bracket, or brace

- `(, [, {` - or immediately after a closing parenthesis, bracket, or brace - `),], }`. The caret is placed before an opening character or after a closing character. If the **CanGoToMatch** property is `$false`, then this method does nothing.

PowerShell

```
# Goes to the matching character if CanGoToMatch() is $true  
$psISE.CurrentPowerShellTab.ConsolePane.GoToMatch()
```

InsertText(text)

Supported in Windows PowerShell ISE 2.0 and later.

Replaces the selection with text or inserts text at the current caret position.

- **text** - String - The text to insert.

See the [Scripting Example](#) later in this topic.

Select(startLine, startColumn, endLine, endColumn)

Supported in Windows PowerShell ISE 2.0 and later.

Selects the text from the **startLine**, **startColumn**, **endLine**, and **endColumn** parameters.

- **startLine** - Integer - The line where the selection starts.
- **startColumn** - Integer - The column within the start line where the selection starts.
- **endLine** - Integer - The line where the selection ends.
- **endColumn** - Integer - The column within the end line where the selection ends.

See the [Scripting Example](#) later in this topic.

SelectCaretLine()

Supported in Windows PowerShell ISE 2.0 and later.

Selects the entire line of text that currently contains the caret.

PowerShell

```
# First, set the caret position on line 5.  
$psISE.CurrentFile.Editor.SetCaretPosition(5,1)  
# Now select that entire line of text  
$psISE.CurrentFile.Editor.SelectCaretLine()
```

SetCaretPosition(lineNumber, columnNumber)

Supported in Windows PowerShell ISE 2.0 and later.

Sets the caret position at the line number and the column number. It throws an exception if either the caret line number or the caret column number are out of their respective valid ranges.

- **lineNumber** - Integer - The caret line number.
- **columnNumber** - Integer - The caret column number.

PowerShell

```
# Set the CaretPosition.  
$psISE.CurrentFile.Editor.SetCaretPosition(5,1)
```

ToggleOutliningExpansion()

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Causes all the outline sections to expand or collapse.

PowerShell

```
# Toggle the outlining expansion  
$psISE.CurrentFile.Editor.ToggleOutliningExpansion()
```

Properties

CanGoToMatch

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

The read-only Boolean property to indicate whether the caret is next to a parenthesis, bracket, or brace - `()`, `[]`, `{}`. If the caret is immediately before the opening character or immediately after the closing character of a pair, then this property value is `$true`. Otherwise, it's `$false`.

PowerShell

```
# Test to see if the caret is next to a parenthesis, bracket, or brace  
$psISE.CurrentFile.Editor.CanGoToMatch
```

CaretColumn

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the column number that corresponds to the position of the caret.

PowerShell

```
# Get the CaretColumn.  
$psISE.CurrentFile.Editor.CaretColumn
```

CaretLine

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the number of the line that contains the caret.

PowerShell

```
# Get the CaretLine.  
$psISE.CurrentFile.Editor.CaretLine
```

CaretLineText

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the complete line of text that contains the caret.

PowerShell

```
# Get all of the text on the line that contains the caret.  
$psISE.CurrentFile.Editor.CaretLineText
```

LineCount

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the line count from the editor.

PowerShell

```
# Get the LineCount.  
$psISE.CurrentFile.Editor.LineCount
```

SelectedText

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the selected text from the editor.

See the [Scripting Example](#) later in this topic.

Text

Supported in Windows PowerShell ISE 2.0 and later.

The read/write property that gets or sets the text in the editor.

See the [Scripting Example](#) later in this topic.

Scripting Example

PowerShell

```
# This illustrates how you can use the length of a line to
# select the entire line and shows how you can make it lowercase.
# You must run this in the Console pane. It will not run in the Script pane.
# Begin by getting a variable that points to the editor.
$myEditor = $psISE.CurrentFile.Editor
# Clear the text in the current file editor.
$myEditor.Clear()

# Make sure the file has five lines of text.
$myEditor.InsertText("LINE1 `n")
$myEditor.InsertText("LINE2 `n")
$myEditor.InsertText("LINE3 `n")
$myEditor.InsertText("LINE4 `n")
$myEditor.InsertText("LINE5 `n")

# Use the GetLineLength method to get the length of the third line.
$endColumn = $myEditor.GetLineLength(3)
# Select the text in the first three lines.
$myEditor.Select(1, 1, 3, $endColumn + 1)
$selection = $myEditor.SelectedText
# Clear all the text in the editor.
$myEditor.Clear()
# Add the selected text back, but in lower case.
$myEditor.InsertText($selection.ToLower())
```

See Also

- [The ISEFile Object](#)
- [The PowerShellTab Object](#)
- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The ISEFileCollection Object

Article • 03/27/2025

The **ISEFileCollection** object is a collection of **ISEFile** objects. An example is the `$psISE.CurrentPowerShellTab.Files` collection.

Methods

Add([FullPath])

Supported in Windows PowerShell ISE 2.0 and later.

Creates and returns a new untitled file and adds it to the collection. The **IsUntitled** property of the newly created file is `$true`.

- **[FullPath]** - Optional string - The fully specified path of the file. An exception is generated if you include the **FullPath** parameter and a relative path, or if you use a file name instead of the full path.

PowerShell

```
# Adds a new untitled file to the collection of files in the current
# PowerShell tab.
$newFile = $psISE.CurrentPowerShellTab.Files.Add()

# Adds a file specified by its full path to the collection of files in the
# current
# PowerShell tab.
$psISE.CurrentPowerShellTab.Files.Add("$PSHOME\Examples\profile.ps1")
```

Remove(File, [Force])

Supported in Windows PowerShell ISE 2.0 and later.

Removes a specified file from the current PowerShell tab.

File - String The **ISEFile** file that you want to remove from the collection. If the file hasn't been saved, this method throws an exception. Use the **Force** switch parameter to force the removal of an unsaved file.

[Force] - optional Boolean If set to `$true`, grants permission to remove the file even if it hasn't been saved after last use. The default is `$false`.

PowerShell

```
# Removes the first opened file from the file collection associated with the
# current
# PowerShell tab. If the file hasn't yet been saved, then an exception is
generated.
$firstfile = $psISE.CurrentPowerShellTab.Files[0]
$psISE.CurrentPowerShellTab.Files.Remove($firstfile)

# Removes the first opened file from the file collection associated with the
# current
# PowerShell tab, even if it hasn't been saved.
$firstfile = $psISE.CurrentPowerShellTab.Files[0]
$psISE.CurrentPowerShellTab.Files.Remove($firstfile, $true)
```

SetSelectedFile(**selectedFile**)

Supported in Windows PowerShell ISE 2.0 and later.

Selects the file that's specified by the **SelectedFile** parameter.

SelectedFile - Microsoft.PowerShell.Host.ISE.ISEFile The ISEFile file that you want to select.

PowerShell

```
# Selects the specified file.
$firstfile = $psISE.CurrentPowerShellTab.Files[0]
$psISE.CurrentPowerShellTab.Files.SetSelectedFile($firstfile)
```

See Also

- [The ISEFile Object](#)
- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The ISEFile Object

Article • 03/27/2025

An **ISEFile** object represents a file in Windows PowerShell Integrated Scripting Environment (ISE). It's an instance of the **Microsoft.PowerShell.Host.ISE.ISEFile** class. This topic lists its member methods and member properties. The `$psISE.CurrentFile` and the files in the `Files` collection in a PowerShell tab are all instances of the **Microsoft.PowerShell.Host.ISE.ISEFile** class.

Methods

Save([saveEncoding])

Supported in Windows PowerShell ISE 2.0 and later.

Saves the file to disk.

`[saveEncoding]` - optional [System.Text.Encoding](#) An optional character encoding parameter to be used for the saved file. The default value is **UTF8**.

Exceptions

- **System.IO.IOException**: The file couldn't be saved.

PowerShell

```
# Save the file using the default encoding (UTF8)
$psISE.CurrentFile.Save()

# Save the file as ASCII.
$psISE.CurrentFile.Save([System.Text.Encoding]::ASCII)

# Gets the current encoding.
$myfile = $psISE.CurrentFile
$myfile.Encoding
```

SaveAs(filename, [saveEncoding])

Supported in Windows PowerShell ISE 2.0 and later.

Saves the file with the specified file name and encoding.

`filename` - String The name to be used to save the file.

`[saveEncoding]` - optional `System.Text.Encoding` An optional character encoding parameter to be used for the saved file. The default value is `UTF8`.

Exceptions

- `System.ArgumentNullException`: The `filename` parameter is null.
- `System.ArgumentException`: The `filename` parameter is empty.
- `System.IO.IOException`: The file couldn't be saved.

PowerShell

```
# Save the file with a full path and name.  
$fullpath = "C:\temp\newname.txt"  
$psISE.CurrentFile.SaveAs($fullPath)  
# Save the file with a full path and name and explicitly as UTF8.  
$psISE.CurrentFile.SaveAs($fullPath, [System.Text.Encoding]::UTF8)
```

Properties

DisplayName

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the string that contains the display name of this file. The name is shown on the **File** tab at the top of the editor. The presence of an asterisk (*) at the end of the name indicates that the file has changes that haven't been saved.

PowerShell

```
# Shows the display name of the file.  
$psISE.CurrentFile.DisplayName
```

Editor

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the [editor object](#) that's used for the specified file.

PowerShell

```
# Gets the editor and the text.  
$psISE.CurrentFile.Editor.Text
```

Encoding

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the original file encoding. This is a **System.Text.Encoding** object.

```
PowerShell  
  
# Shows the encoding for the file.  
$psISE.CurrentFile.Encoding
```

FullPath

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the string that specifies the full path of the opened file.

```
PowerShell  
  
# Shows the full path for the file.  
$psISE.CurrentFile.FullPath
```

IsSaved

Supported in Windows PowerShell ISE 2.0 and later.

The read-only Boolean property that returns `$true` if the file has been saved after it was last modified.

```
PowerShell  
  
# Determines whether the file has been saved since it was last modified.  
$myfile = $psISE.CurrentFile  
$myfile.IsSaved
```

IsUntitled

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that returns `$true` if the file has never been given a title.

PowerShell

```
# Determines whether the file has never been given a title.  
$psISE.CurrentFile.IsUntitled  
$psISE.CurrentFile.SaveAs("temp.txt")  
$psISE.CurrentFile.IsUntitled
```

See Also

- [The ISEFileCollectionObject](#)
- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The ISEMenuItemCollection Object

Article • 03/27/2025

An **ISEMenuItemCollection** object is a collection of **ISEMenuItem** objects. It's an instance of the **Microsoft.PowerShell.Host.ISE.ISEMenuItemCollection** class. An example is the `$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus` object that's used to customize the **Add-On** menu in Windows PowerShell® Integrated Scripting Environment (ISE).

Method

Add(**DisplayName**, **Action**, **Shortcut**)

Supported in Windows PowerShell ISE 2.0 and later.

Adds a menu item to the collection.

- **DisplayName** - String - The display name of the menu to be added.
- **Action** - System.Management.Automation.ScriptBlock - The object that specifies the action that's associated with this menu item.
- **Shortcut** - System.Windows.Input.KeyGesture - The keyboard shortcut for the action.
- **Returns** - The **ISEMenuItem** object that was just added.

PowerShell

```
# Create an Add-ons menu with a fast access key and a shortcut.
# Note the use of "_" as opposed to the "&" for mapping to the fast access
key
# letter for the menu item.
$menuAdded = $psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Add('_Process',
    {Get-Process}, 'Alt+P')
```

Clear()

Supported in Windows PowerShell ISE 2.0 and later.

Removes all submenus from the menu item.

PowerShell

```
# Remove all custom submenu items from the AddOns menu
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Clear()
```

See Also

- [The ISEMenuItem Object](#)
- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The ISEMenuItem Object

Article • 03/27/2025

An **ISEMenuItem** object is an instance of the **Microsoft.PowerShell.Host.ISE.ISEMenuItem** class. All menu objects on the **Add-ons** menu are instances of the **Microsoft.PowerShell.Host.ISE.ISEMenuItem** class.

Properties

DisplayName

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the display name of the menu item.

PowerShell

```
# Get the display name of the Add-ons menu item
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Clear()
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Add('_Process', {Get-
    Process}, 'Alt+P')
$psISE.CurrentPowerShellTab.AddOnsMenu.DisplayName
```

Action

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the block of script. It invokes the action when you click the menu item.

PowerShell

```
# Get the action associated with the first submenu item.
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Clear()
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Add('_Process', {Get-
    Process}, 'Alt+P')
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus[0].Action

# Invoke the script associated with the first submenu item
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus[0].Action.Invoke()
```

Shortcut

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the Windows input keyboard shortcut for the menu item.

PowerShell

```
# Get the shortcut for the first submenu item.  
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Clear()  
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Add('_Process', {Get-  
Process}, 'Alt+P')  
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus[0].Shortcut
```

Submenus

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the [list of submenus](#) of the menu item.

PowerShell

```
# List the submenus of the Add-ons menu  
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Clear()  
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Add('_Process', {Get-  
Process}, 'Alt+P')  
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus
```

Scripting example

To better understand the use of the Add-ons menu and its scriptable properties, read through the following scripting example.

PowerShell

```
# This is a scripting example that shows the use of the Add-ons menu.  
# Clear the Add-ons menu if any entries currently exist  
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Clear()  
  
# Add an Add-ons menu item with a shortcut and fast access key.  
# Note the use of "_" as opposed to the "&" for mapping to the fast access  
key letter  
# for the menu item.  
$menuAdded = $psISE.CurrentPowerShellTab.AddOnsMenu.SubMenus.Add('_Process',  
{Get-Process}, 'Alt+P')  
# Add a nested menu - a parent and a child submenu item.  
$parentAdded = $psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Add('Parent',
```

```
$null, $null)  
$parentAdded.SubMenus.Add('_Dir', {dir}, 'Alt+D')
```

See Also

- [The ISEMenuItemCollection Object](#)
- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The ISEOptions Object

Article • 04/30/2025

The **ISEOptions** object represents various settings for Windows PowerShell ISE. It's an instance of the **Microsoft.PowerShell.Host.ISE.ISEOptions** class.

The **ISEOptions** object provides the following methods and properties.

Methods

RestoreDefaultConsoleTokenColors()

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Restores the default values of the token colors in the Console pane.

PowerShell

```
# Changes the color of the commands in the Console pane to red and then restores
# it to its default value.
$psISE.Options.ConsoleTokenColors["Command"] = 'red'
$psISE.Options.RestoreDefaultConsoleTokenColors()
```

RestoreDefaults()

Supported in Windows PowerShell ISE 2.0 and later.

Restores the default values of all options settings in the Console pane. It also resets the behavior of various warning messages that provide the standard check box to prevent the message from being shown again.

PowerShell

```
# Changes the background color in the Console pane and then restores it to its
# default value.
$psISE.Options.ConsolePaneBackgroundColor = 'orange'
$psISE.Options.RestoreDefaults()
```

RestoreDefaultTokenColors()

Supported in Windows PowerShell ISE 2.0 and later.

Restores the default values of the token colors in the Script pane.

PowerShell

```
# Changes the color of the comments in the Script pane to red and then restores it
# to its default value.
$psISE.Options.TokenColors["Comment"] = 'red'
$psISE.Options.RestoreDefaultTokenColors()
```

RestoreDefaultXmlTokenColors()

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Restores the default values of the token colors for XML elements that are displayed in Windows PowerShell ISE. Also see [XmlAttributeColors](#).

PowerShell

```
# Changes the color of the comments in XML data to red and then restores it
# to its default value.
$psISE.Options.XmlTokenColors["Comment"] = 'red'
$psISE.Options.RestoreDefaultXmlTokenColors()
```

Properties

AutoSaveMinuteInterval

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies the number of minutes between automatic save operations of your files by Windows PowerShell ISE. The default value is 2 minutes. The value is an integer.

PowerShell

```
# Changes the number of minutes between automatic save operations to every 3
minutes.
$psISE.Options.AutoSaveMinuteInterval = 3
```

CommandPaneBackgroundColor

This feature is present in Windows PowerShell ISE 2.0, but was removed or renamed in later versions of the ISE. For later versions, see [ConsolePaneBackgroundColor](#).

Specifies the background color for the Command pane. It's an instance of the **System.Windows.Media.Color** class.

PowerShell

```
# Changes the background color of the Command pane to orange.  
$psISE.Options.CommandPaneBackgroundColor = 'orange'
```

CommandPaneUp

This feature is present in Windows PowerShell ISE 2.0, but was removed or renamed in later versions of the ISE.

Specifies whether the Command pane is located above the Output pane.

PowerShell

```
# Moves the Command pane to the top of the screen.  
$psISE.Options.CommandPaneUp = $true
```

ConsolePaneBackgroundColor

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies the background color for the Console pane. It's an instance of the **System.Windows.Media.Color** class.

PowerShell

```
# Changes the background color of the Console pane to red.  
$psISE.Options.ConsolePaneBackgroundColor = 'red'
```

ConsolePaneForegroundColor

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies the foreground color of the text in the Console pane.

PowerShell

```
# Changes the foreground color of the text in the Console pane to yellow.  
$psISE.Options.ConsolePaneForegroundColor = 'yellow'
```

ConsolePaneTextColorBackgroundColor

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies the background color of the text in the Console pane.

PowerShell

```
# Changes the background color of the Console pane text to pink.  
$psISE.Options.ConsolePaneTextColorBackgroundColor = 'pink'
```

ConsoleTokenColors

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies the colors of the IntelliSense tokens in the Windows PowerShell ISE Console pane. This property is a dictionary object that contains name/value pairs of token types and colors for the Console pane. To change the colors of the IntelliSense tokens in the Script pane, see [TokenColors](#). To reset the colors to the default values, see [RestoreDefaultConsoleTokenColors](#). Token colors can be set for the following: `Attribute`, `Command`, `CommandArgument`, `CommandParameter`, `Comment`, `GroupEnd`, `GroupStart`, `Keyword`, `LineContinuation`, `LoopLabel`, `Member`, `NewLine`, `Number`, `Operator`, `Position`, `StatementSeparator`, `String`, `Type`, `Unknown`, `Variable`.

PowerShell

```
# Sets the color of commands to green.  
$psISE.Options.ConsoleTokenColors[ "Command" ] = 'green'  
# Sets the color of keywords to magenta.  
$psISE.Options.ConsoleTokenColors[ "Keyword" ] = 'magenta'
```

DebugBackgroundColor

Supported in Windows PowerShell ISE 2.0 and later.

Specifies the background color for the debug text that appears in the Console pane. It's an instance of the `System.Windows.Media.Color` class.

PowerShell

```
# Changes the background color for the debug text that appears in the Console pane  
# to blue.  
$psISE.Options.DebugBackgroundColor = '#0000FF'
```

DebugForegroundColor

Supported in Windows PowerShell ISE 2.0 and later.

Specifies the foreground color for the debug text that appears in the Console pane. It's an instance of the `System.Windows.Media.Color` class.

PowerShell

```
# Changes the foreground color for the debug text that appears in the Console
# pane to yellow.
$psISE.Options.DebugForegroundColor = 'yellow'
```

DefaultOptions

Supported in Windows PowerShell ISE 2.0 and later.

A collection of properties that specify the default values to be used when the Reset methods are used.

PowerShell

```
# Displays the name of the default options. This example is from ISE 4.0.
$psISE.Options.DefaultOptions
```

Output

```
SelectedScriptPaneState : Top
ShowDefaultSnippets : True
ShowToolBar : True
ShowOutlining : True
ShowLineNumbers : True
TokenColors : {[Attribute, #FF00BFFF], [Command,
#FF0000FF],
[CommandParameter, #FF000080]...}
ConsoleTokenColors : {[Attribute, #FFB0C4DE], [Command,
#FFE0FFFF],
[CommandParameter, #FFFFE4B5]...}
XmlTokenColors : {[Comment, #FF006400],
[ElementName, #FF8B0000],
[MarkupExtension, #FFFF8C00]...}
DefaultOptions :
Microsoft.PowerShell.Host.ISE.ISEOptions
FontSize : 9
Zoom : 100
```

```
FontName : Lucida Console
ErrorForegroundColor : #FFFF9494
ErrorBackgroundColor : #00FFFFFF
WarningForegroundColor : #FFFF8C00
WarningBackgroundColor : #00FFFFFF
VerboseForegroundColor : #FF00FFFF
VerboseBackgroundColor : #00FFFFFF
DebugForegroundColor : #FF00FFFF
DebugBackgroundColor : #00FFFFFF
ConsolePaneBackgroundColor : #FF012456
ConsolePaneTextBackgroundColor : #FF012456
ConsolePaneForegroundColor : #FFF5F5F5
ScriptPaneBackgroundColor : #FFFFFFFF
ScriptPaneForegroundColor : #FF000000
ShowWarningForDuplicateFiles : True
ShowWarningBeforeSavingOnRun : True
UseLocalHelp : True
AutoSaveMinuteInterval : 2
Mrucount : 10
ShowIntellisenseInConsolePane : True
ShowIntellisenseInScriptPane : True
UseEnterToSelectInConsolePaneIntellisense : True
UseEnterToSelectInScriptPaneIntellisense : True
```

ErrorBackgroundColor

Supported in Windows PowerShell ISE 2.0 and later.

Specifies the background color for error text that appears in the Console pane. It's an instance of the [System.Windows.Media.Color](#) class.

PowerShell

```
# Changes the background color for the error text that appears in the Console pane
# to black.
$psISE.Options.ErrorBackgroundColor = 'black'
```

ErrorForegroundColor

Supported in Windows PowerShell ISE 2.0 and later.

Specifies the foreground color for error text that appears in the Console pane. It's an instance of the [System.Windows.Media.Color](#) class.

PowerShell

```
# Changes the foreground color for the error text that appears in the console pane
# to green.
```

```
$psISE.Options.ErrorForegroundColor = 'green'
```

FontName

Supported in Windows PowerShell ISE 2.0 and later.

Specifies the font name currently in use in both the Script pane and the Console pane.

PowerShell

```
# Changes the font used in both panes.  
$psISE.Options.FontName = 'Courier New'
```

FontSize

Supported in Windows PowerShell ISE 2.0 and later.

Specifies the font size as an integer. It's used in the Script pane, the Command pane, and the Output pane. The valid range of values is 8 through 32.

PowerShell

```
# Changes the font size in all panes.  
$psISE.Options.FontSize = 20
```

IntellisenseTimeoutInSeconds

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies the number of seconds that IntelliSense uses to try to resolve the currently typed text. After this number of seconds, IntelliSense times out and enables you to continue typing. The default value is 3 seconds. The value is an integer.

PowerShell

```
# Changes the number of seconds for IntelliSense syntax recognition to 5.  
$psISE.Options.IntellisenseTimeoutInSeconds = 5
```

MruCount

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies the number of recently opened files that Windows PowerShell ISE tracks and displays at the bottom of the **File Open** menu. The default value is 10. The value is an integer.

PowerShell

```
# Changes the number of recently used files that appear at the bottom of the
# File Open menu to 5.
$psISE.Options.MruCount = 5
```

OutputPaneBackgroundColor

This feature is present in Windows PowerShell ISE 2.0, but was removed or renamed in later versions of the ISE. For later versions, see [ConsolePaneBackgroundColor](#).

The read/write property that gets or sets the background color for the Output pane itself. It's an instance of the **System.Windows.Media.Color** class.

PowerShell

```
# Changes the background color of the Output pane to gold.
$psISE.Options.OutputPaneBackgroundColor = 'gold'
```

OutputPaneTextForegroundColor

This feature is present in Windows PowerShell ISE 2.0, but was removed or renamed in later versions of the ISE. For later versions, see [ConsolePaneForegroundColor](#).

The read/write property that changes the foreground color of the text in the Output pane in Windows PowerShell ISE 2.0.

PowerShell

```
# Changes the foreground color of the text in the Output Pane to blue.
$psISE.Options.OutputPaneTextForegroundColor = 'blue'
```

OutputPaneTextBackgroundColor

This feature is present in Windows PowerShell ISE 2.0, but was removed or renamed in later versions of the ISE. For later versions, see [ConsolePaneTextBackgroundColor](#).

The read/write property that changes the background color of the text in the Output pane.

PowerShell

```
# Changes the background color of the Output pane text to pink.  
$psISE.Options.OutputPaneTextColor = 'pink'
```

ScriptPaneBackgroundColor

Supported in Windows PowerShell ISE 2.0 and later.

The read/write property that gets or sets the background color for files. It's an instance of the **System.Windows.Media.Color** class.

PowerShell

```
# Sets the color of the script pane background to yellow.  
$psISE.Options.ScriptPaneBackgroundColor = 'yellow'
```

ScriptPaneForegroundColor

Supported in Windows PowerShell ISE 2.0 and later.

The read/write property that gets or sets the foreground color for non-script files in the Script pane. To set the foreground color for script files, use the [TokenColors](#).

PowerShell

```
# Sets the foreground to color of non-script files in the script pane to green.  
$psISE.Options.ScriptPaneTextColor = 'green'
```

SelectedScriptPaneState

Supported in Windows PowerShell ISE 2.0 and later.

The read/write property that gets or sets the position of the Script pane on the display. The string can be either 'Maximized', 'Top', or 'Right'.

PowerShell

```
# Moves the Script Pane to the top.  
$psISE.Options.SelectedScriptPaneState = 'Top'  
# Moves the Script Pane to the right.  
$psISE.Options.SelectedScriptPaneState = 'Right'  
# Maximizes the Script Pane  
$psISE.Options.SelectedScriptPaneState = 'Maximized'
```

ShowDefaultSnippets

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies whether the `Ctrl + J` list of snippets includes the starter set that's included in Windows PowerShell. When set to `$false`, only user-defined snippets appear in the `Ctrl + J` list. The default value is `$true`.

PowerShell

```
# Hide the default snippets from the Ctrl+J list.  
$psISE.Options.ShowDefaultSnippets = $false
```

ShowIntellisenseInConsolePane

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies whether IntelliSense offers syntax, parameter, and value suggestions in the Console pane. The default value is `$true`.

PowerShell

```
# Turn off IntelliSense in the console pane.  
$psISE.Options.ShowIntellisenseInConsolePane = $false
```

ShowIntellisenseInScriptPane

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies whether IntelliSense offers syntax, parameter, and value suggestions in the Script pane. The default value is `$true`.

PowerShell

```
# Turn off IntelliSense in the Script pane.  
$psISE.Options.ShowIntellisenseInScriptPane = $false
```

ShowLineNumbers

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies whether the Script pane displays line numbers in the left margin. The default value is \$true.

PowerShell

```
# Turn off line numbers in the Script pane.  
$psISE.Options.ShowLineNumbers = $false
```

ShowOutlining

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies whether the Script pane displays expandable and collapsible brackets next to sections of code in the left margin. When they're displayed, you can click the minus - icons next to a block of text to collapse it or click the plus + icon to expand a block of text. The default value is \$true.

PowerShell

```
# Turn off outlining in the Script pane.  
$psISE.Options.ShowOutlining = $false
```

ShowToolBar

Supported in Windows PowerShell ISE 2.0 and later.

Specifies whether the ISE toolbar appears at the top of the Windows PowerShell ISE window. The default value is \$true.

PowerShell

```
# Show the toolbar.  
$psISE.Options.ShowToolBar = $true
```

ShowWarningBeforeSavingOnRun

Supported in Windows PowerShell ISE 2.0 and later.

Specifies whether a warning message appears when a script is saved automatically before it's run. The default value is \$true.

PowerShell

```
# Enable the warning message when an attempt
# is made to run a script without saving it first.
$psISE.Options.ShowWarningBeforeSavingOnRun = $true
```

ShowWarningForDuplicateFiles

Supported in Windows PowerShell ISE 2.0 and later.

Specifies whether a warning message appears when the same file is opened in different PowerShell tabs. If set to `$true`, to open the same file in multiple tabs displays this message: "A copy of this file is open in another Windows PowerShell tab. Changes made to this file will affect all open copies." The default value is `$true`.

PowerShell

```
# Enable the warning message when a file is
# opened in multiple PowerShell tabs.
$psISE.Options.ShowWarningForDuplicateFiles = $true
```

TokenColors

Supported in Windows PowerShell ISE 2.0 and later.

Specifies the colors of the IntelliSense tokens in the Windows PowerShell ISE Script pane. This property is a dictionary object that contains name/value pairs of token types and colors for the Script pane. To change the colors of the IntelliSense tokens in the Console pane, see [ConsoleTokenColors](#). To reset the colors to the default values, see [RestoreDefaultTokenColors](#). Token colors can be set for the following: Attribute, Command, CommandArgument, CommandParameter, Comment, GroupEnd, GroupStart, Keyword, LineContinuation, LoopLabel, Member, NewLine, Number, Operator, Position, StatementSeparator, String, Type, Unknown, Variable.

PowerShell

```
# Sets the color of commands to green.
$psISE.Options.TokenColors["Command"] = "green"
# Sets the color of keywords to magenta.
$psISE.Options.TokenColors["Keyword"] = "magenta"
```

UseEnterToSelectInConsolePanelIntellisense

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies whether you can use the Enter key to select an IntelliSense provided option in the Console pane. The default value is `$true`.

PowerShell

```
# Turn off using the ENTER key to select an IntelliSense provided option in the
# Console pane.
$psISE.Options.UseEnterToSelectInConsolePaneIntellisense = $false
```

UseEnterToSelectInScriptPanelIntellisense

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies whether you can use the Enter key to select an IntelliSense-provided option in the Script pane. The default value is `$true`.

PowerShell

```
# Turn on using the Enter key to select an IntelliSense provided option in the
# Console pane.
$psISE.Options.UseEnterToSelectInConsolePaneIntellisense = $true
```

UseLocalHelp

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies whether the locally installed Help or the online Help appears when you press `F1` with the cursor positioned in a keyword. If set to `$true`, then a pop-up window shows content from the locally installed Help. You can install the Help files by running the `Update-Help` command. If set to `$false`, then your browser opens to a page on Microsoft Learn.

PowerShell

```
# Sets the option for the online help to be displayed.
$psISE.Options.UseLocalHelp = $false
# Sets the option for the local Help to be displayed.
$psISE.Options.UseLocalHelp = $true
```

VerboseBackgroundColor

Supported in Windows PowerShell ISE 2.0 and later.

Specifies the background color for verbose text that appears in the Console pane. It's a **System.Windows.Media.Color** object.

PowerShell

```
# Changes the background color for verbose text to blue.  
$psISE.Options.VerboseBackgroundColor = '#0000FF'
```

VerboseForegroundColor

Supported in Windows PowerShell ISE 2.0 and later.

Specifies the foreground color for verbose text that appears in the Console pane. It's a **System.Windows.Media.Color** object.

PowerShell

```
# Changes the foreground color for verbose text to yellow.  
$psISE.Options.VerboseForegroundColor = 'yellow'
```

WarningBackgroundColor

Supported in Windows PowerShell ISE 2.0 and later.

Specifies the background color for warning text that appears in the Console pane. It's a **System.Windows.Media.Color** object.

PowerShell

```
# Changes the background color for warning text to blue.  
$psISE.Options.WarningBackgroundColor = '#0000FF'
```

WarningForegroundColor

Supported in Windows PowerShell ISE 2.0 and later.

Specifies the foreground color for warning text that appears in the Output pane. It's a **System.Windows.Media.Color** object.

PowerShell

```
# Changes the foreground color for warning text to yellow.
```

```
$psISE.Options.WarningForegroundColor = 'yellow'
```

XmlTokenColors

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies a dictionary object that contains name/value pairs of token types and colors for XML content that's displayed in Windows PowerShell ISE. Token colors can be set for the following: Attribute, Command, CommandArgument, CommandParameter, Comment, GroupEnd, GroupStart, Keyword, LineContinuation, LoopLabel, Member, NewLine, Number, Operator, Position, StatementSeparator, String, Type, Unknown, Variable. Also see [RestoreDefaultXmlTokenColors](#).

PowerShell

```
# Sets the color of XML element names to green.  
$psISE.Options.XmlTokenColors["ElementName"] = 'green'  
# Sets the color of XML comments to magenta.  
$psISE.Options.XmlTokenColors["Comment"] = 'magenta'
```

Zoom

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Specifies the relative size of text in both the Console and Script panes. The default value is 100. Smaller values cause the text in Windows PowerShell ISE to appear smaller while larger numbers cause text to appear larger. The value is an integer that ranges from 20 to 400.

PowerShell

```
# Changes the text in the Windows PowerShell ISE to be double its normal size.  
$psISE.Options.Zoom = 200
```

See Also

- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The ISESnippetCollection Object

Article • 03/27/2025

The **ISESnippetCollection** object is a collection of **ISESnippet** objects. The files collection that's associated with a **PowerShellTab** object is a member of this class. An example is the `$psISE.CurrentPowerShellTab.Files` collection.

Methods

Load(FilePathName)

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Loads a `snippets.ps1xml` file that contains user-defined snippets. The easiest way to create snippets is to use the `New-IseSnippet` cmdlet, which automatically stores them in your profile folder so that they're loaded every time that you start Windows PowerShell ISE.

- **FilePathName** - String - The path and file name to a `snippets.ps1xml` file that contains snippet definitions.

PowerShell

```
# Loads a custom snippet file into the current PowerShell tab.
$joinPathSplat = @{
    Path = ( Split-Path $PROFILE)
    ChildPath = 'Snippets\MySnips.snippets.ps1xml'
    AdditionalChildPath =
}
$snipPath = Join-Path @joinPathSplat
```

See Also

- [The ISESnippetObject](#)
- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The ISESnippetObject

Article • 03/27/2025

An **ISESnippet** object is an instance of the `Microsoft.PowerShell.Host.ISE.ISESnippet` class. The members of the `$psISE.CurrentPowerShellTab.Snippets` collection are all examples of **ISESnippet** objects. The easiest way to create a snippet is to use the [New-IseSnippet](#) cmdlet.

Properties

Author

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

The read-only property that gets the name of the author of the snippet.

PowerShell

```
# Get the author of the first snippet item
$psISE.CurrentPowerShellTab.Snippets.Item(0).Author
```

CodeFragment

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

The read-only property that gets the code fragment to be inserted into the editor.

PowerShell

```
# Get the code fragment associated with the first snippet item.
$psISE.CurrentPowerShellTab.Snippets.Item(0).CodeFragment
```

Shortcut

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

The read-only property that gets the Windows keyboard shortcut for the menu item.

PowerShell

```
# Get the shortcut for the first submenu item.  
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Clear()  
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus.Add('_Process', {Get-  
Process}, 'Alt+P')  
$psISE.CurrentPowerShellTab.AddOnsMenu.Submenus[0].Shortcut
```

See Also

- [The ISESnippetCollection Object](#)
- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The PowerShellTabCollection Object

Article • 03/27/2025

The **PowerShellTab** collection object is a collection of **PowerShellTab** objects. Each **PowerShellTab** object functions as a separate runtime environment. It's an instance of Microsoft.PowerShell.Host.ISE.PowerShellTabs class. An example is the `$psISE.PowerShellTabs` object.

Methods

Add()

Supported in Windows PowerShell ISE 2.0 and later.

Adds a new PowerShell tab to the collection. It returns the newly added tab.

PowerShell

```
$newTab = $psISE.PowerShellTabs.Add()  
$newTab.DisplayName = 'Brand New Tab'
```

Remove(**Microsoft.PowerShell.Host.ISE.PowerShellTab** **psTab**)

Supported in Windows PowerShell ISE 2.0 and later.

Removes the tab that's specified by the **psTab** parameter.

- **psTab** - The PowerShell tab to remove.

PowerShell

```
$newTab = $psISE.PowerShellTabs.Add()  
Change the DisplayName of the new PowerShell tab.  
$newTab.DisplayName = 'This tab will go away in 5 seconds'  
sleep 5  
$psISE.PowerShellTabs.Remove($newTab)
```

SetSelectedPowerShellTab(**Microsoft.PowerShell.Host.ISE.** **PowerShellTab** **psTab**)

Supported in Windows PowerShell ISE 2.0 and later.

Selects the PowerShell tab that's specified by the **psTab** parameter to make it the currently active PowerShell tab.

- **psTab** - The PowerShell tab to select.

```
PowerShell

# Save the current tab in a variable and rename it
$oldTab = $psISE.CurrentPowerShellTab
$psISE.CurrentPowerShellTab.DisplayName = 'Old Tab'
# Create a new tab and give it a new display name
$newTab = $psISE.PowerShellTabs.Add()
$newTab.DisplayName = 'Brand New Tab'
# Switch back to the original tab
$psISE.PowerShellTabs.SelectedPowerShellTab = $oldTab
```

See Also

- [The PowerShellTab Object](#)
- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

The PowerShellTab Object

Article • 03/27/2025

The **PowerShellTab** object represents a Windows PowerShell runtime environment.

Methods

Invoke(Script)

Supported in Windows PowerShell ISE 2.0 and later.

Runs the given script in the PowerShell tab.

ⓘ Note

This method only works on other PowerShell tabs, not the PowerShell tab from which it's run. It doesn't return any object or value. If the code modifies any variable, then those changes persist on the tab against which the command was invoked.

- **Script** - System.Management.Automation.ScriptBlock or String - The script block to run.

PowerShell

```
# Manually create a second PowerShell tab before running this script.  
# Return to the first PowerShell tab and type the following command  
$psISE.PowerShellTabs[1].Invoke({dir})
```

InvokeSynchronous(Script, [useNewScope], millisecondsTimeout)

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

Runs the given script in the PowerShell tab.

ⓘ Note

This method only works on other PowerShell tabs, not the PowerShell tab from which it's run. The script block is run and any value that's returned from the script is

returned to the run environment from which you invoked the command. If the command takes longer to run than the **millisecondsTimeout** value specifies, then the command fails with an exception: "The operation has timed out."

- **Script** - System.Management.Automation.ScriptBlock or String - The script block to run.
- **[useNewScope]** - Optional Boolean that defaults to `$true` - If set to `$true`, then a new scope is created within which to run the command. It doesn't modify the runtime environment of the PowerShell tab that's specified by the command.
- **[millisecondsTimeout]** - Optional integer that defaults to `500`. - If the command doesn't finish within the specified time, then the command generates a **TimeoutException** with the message "The operation has timed out."

PowerShell

```
# Create a new PowerShell tab and then switch back to the first
$psISE.PowerShellTabs.Add()
$psISE.PowerShellTabs.setSelectedPowerShellTab($psISE.PowerShellTabs[0])

# Invoke a simple command on the other tab, in its own scope
$psISE.PowerShellTabs[1].InvokeSynchronous('$x=1', $false)
# You can switch to the other tab and type '$x' to see that the value is
# saved there.

# This example sets a value in the other tab (in a different scope)
# and returns it through the pipeline to this tab to store in $a
$a = $psISE.PowerShellTabs[1].InvokeSynchronous('$z=3;$z')

# This example runs a command that takes longer than the allowed timeout
# value
# and measures how long it runs so that you can see the impact
Measure-Command {$psISE.PowerShellTabs[1].InvokeSynchronous('sleep 10',
    $false, 5000)}
```

Properties

AddOnsMenu

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the Add-ons menu for the PowerShell tab.

PowerShell

```
# Clear the Add-ons menu if one exists.  
$psISE.CurrentPowerShellTab.AddOnsMenu.SubMenus.Clear()  
# Create an AddOns menu with an accessor.  
# Note the use of "_" as opposed to the "&" for mapping to the fast key  
letter for the menu item.  
$menuAdded = $psISE.CurrentPowerShellTab.AddOnsMenu.SubMenus.Add('Process',  
    {Get-Process}, 'Alt+P')  
# Add a nested menu.  
$parentAdded = $psISE.CurrentPowerShellTab.AddOnsMenu.SubMenus.Add('Parent',  
    $null, $null)  
$parentAdded.SubMenus.Add('Dir', {dir}, 'Alt+D')  
# Show the Add-ons menu on the current PowerShell tab.  
$psISE.CurrentPowerShellTab.AddOnsMenu
```

CanInvoke

Supported in Windows PowerShell ISE 2.0 and later.

The read-only Boolean property that returns a `$true` value if a script can be invoked with the [Invoke\(Script \)](#) method.

PowerShell

```
# CanInvoke will be false if the PowerShell  
# tab is running a script that takes a while, and you  
# check its properties from another PowerShell tab. It's  
# always false if checked on the current PowerShell tab.  
# Manually create a second PowerShell tab before running this script.  
# Return to the first tab and type  
$secondTab = $psISE.PowerShellTabs[1]  
$secondTab.CanInvoke  
$secondTab.Invoke({sleep 20})  
$secondTab.CanInvoke
```

ConsolePane

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions. In Windows PowerShell ISE 2.0 this was named **CommandPane**.

The read-only property that gets the Console pane [editor](#) object.

PowerShell

```
# Gets the Console Pane editor.  
$psISE.CurrentPowerShellTab.ConsolePane
```

DisplayName

Supported in Windows PowerShell ISE 2.0 and later.

The read-write property that gets or sets the text that's displayed on the PowerShell tab. By default, tabs are named "PowerShell #", where the # represents a number.

PowerShell

```
$newTab = $psISE.PowerShellTabs.Add()  
# Change the DisplayName of the new PowerShell tab.  
$newTab.DisplayName = 'Brand New Tab'
```

ExpandedScript

Supported in Windows PowerShell ISE 2.0 and later.

The read-write Boolean property that determines whether the Script pane is expanded or hidden.

PowerShell

```
# Toggle the expanded script property to see its effect.  
$psISE.CurrentPowerShellTab.ExpandedScript =  
!$psISE.CurrentPowerShellTab.ExpandedScript
```

Files

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the [collection of script files](#) that are open in the PowerShell tab.

PowerShell

```
$newFile = $psISE.CurrentPowerShellTab.Files.Add()  
$newFile.Editor.Text = "a`r`nb"  
# Gets the line count  
$newFile.Editor.LineCount
```

Output

This feature is present in Windows PowerShell ISE 2.0, but was removed or renamed in later versions of the ISE. In later versions of Windows PowerShell ISE, you can use the **ConsolePane** object for the same purposes.

The read-only property that gets the Output pane of the current [editor](#).

```
PowerShell

# Clears the text in the Output pane.
$psISE.CurrentPowerShellTab.Output.Clear()
```

Prompt

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the current prompt text. Note: the `prompt` function can be overridden by the user's profile. If the result is other than a simple string, then this property returns nothing.

```
PowerShell

# Gets the current prompt text.
$psISE.CurrentPowerShellTab.Prompt
```

ShowCommands

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

The read-write property that indicates if the Commands pane is currently displayed.

```
PowerShell

# Gets the current status of the Commands pane and stores it in the $a
variable
$a = $psISE.CurrentPowerShellTab.ShowCommands
# if $a is $false, then turn the Commands pane on by changing the value to
$true
if (!$a) {$psISE.CurrentPowerShellTab.ShowCommands = $true}
```

StatusText

Supported in Windows PowerShell ISE 2.0 and later.

The read-only property that gets the **PowerShellTab** status text.

PowerShell

```
# Gets the current status text,  
$psISE.CurrentPowerShellTab.StatusText
```

HorizontalAddOnToolsPaneOpened

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

The read-only property that indicates whether the horizontal Add-Ons tool pane is currently open.

PowerShell

```
# Gets the current state of the horizontal Add-ons tool pane.  
$psISE.CurrentPowerShellTab.HorizontalAddOnToolsPaneOpened
```

VerticalAddOnToolsPaneOpened

Supported in Windows PowerShell ISE 3.0 and later, and not present in earlier versions.

The read-only property that indicates whether the vertical Add-Ons tool pane is currently open.

PowerShell

```
# Turns on the Commands pane  
$psISE.CurrentPowerShellTab.ShowCommands = $true  
# Gets the current state of the vertical Add-ons tool pane.  
$psISE.CurrentPowerShellTab.VerticalAddOnToolsPaneOpened
```

See Also

- [The PowerShellTabCollection Object](#)
- [Purpose of the Windows PowerShell ISE Scripting Object Model](#)
- [The ISE Object Model Hierarchy](#)

Other Useful Scripting Objects

Article • 03/27/2025

The following objects provide additional scripting functionality in Windows PowerShell ISE. They aren't part of the `$psISE` hierarchy.

Useful Scripting objects

\$psUnsupportedConsoleApplications

There are some limitations on how Windows PowerShell ISE interacts with console applications. A command or an automation script that requires user intervention might not work the way it works from the Windows PowerShell console. You might want to block these commands or scripts from running in the Windows PowerShell ISE Command pane. The `$psUnsupportedConsoleApplications` object keeps a list of such commands. If you try to run the commands in this list, you get a message that they aren't supported. The following script adds an entry to the list.

```
PowerShell

# List the unsupported commands
$psUnsupportedConsoleApplications

# Add a command to this list
$psUnsupportedConsoleApplications.Add('Mycommand')

# Show the augmented list of commands
$psUnsupportedConsoleApplications
```

\$psLocalHelp

This is a dictionary object that maintains a context-sensitive mapping between Help topics and their associated links in the local compiled HTML Help file. It's used to locate the local Help for a particular topic. You can add or delete topics from this list. The following code example shows some example key-value pairs that are contained in `$psLocalHelp`.

```
PowerShell

# See the local help map
$psLocalHelp | Format-List
```

Output

```
Key    : Add-Computer
Value  : WindowsPowerShellHelp.chm::/html/093f660c-b8d5-43cf-aa0c-
54e5e54e76f9.htm
```

```
Key    : Add-Content
Value  : WindowsPowerShellHelp.chm::/html/0c836a1b-f389-4e9a-9325-
0f415686d194.htm
```

The following script adds an entry to the list.

PowerShell

```
$psLocalHelp.Add("Get-MyNoun", "C:\MyFolder\MyHelpChm.chm::/html/0198854a-
1298-57ae-aa0c-87b5e5a84712.htm")
```

\$psOnlineHelp

This is a dictionary object that maintains a context-sensitive mapping between topic titles of Help topics and their associated external URLs. It's used to locate the Help for a particular topic on the web. You can add or delete topics from this list.

PowerShell

```
$psOnlineHelp | Format-List
```

Output

```
Key    : Add-Computer
Value  : https://go.microsoft.com/fwlink/?LinkID=135194
```

```
Key    : Add-Content
Value  : https://go.microsoft.com/fwlink/?LinkID=113278
```

The following script adds an entry to the list.

PowerShell

```
$psOnlineHelp.Add("Get-MyNoun", "https://www.mydomain.com/MyNoun.html")
```

See Also

Purpose of the Windows PowerShell ISE Scripting Object Model

Starting Windows PowerShell

Article • 03/27/2025

Windows PowerShell is a scripting engine embedded into multiple hosts. The most common hosts are the interactive command-line `powershell.exe` and the Interactive Scripting Environment `powershell_ise.exe`.

PowerShell binary name

PowerShell version 6 and higher uses .NET (Core). Supported versions are available on Windows, macOS, and Linux.

Beginning in PowerShell 6, the PowerShell binary named `pwsh.exe` for Windows and `pwsh` for macOS and Linux. You can start PowerShell preview versions using `pwsh-preview`. For more information, see [About pwsh](#).

To find cmdlet reference and installation documentation for PowerShell 7, use the following links:

Expand table

Document	Link
Cmdlet reference	PowerShell Module Browser
Windows installation	Installing PowerShell on Windows
macOS installation	Installing PowerShell on macOS
Linux installation	Installing PowerShell on Linux

To view content for other PowerShell versions, see [How to use the PowerShell documentation](#).

Run from the Start Menu

- Open the Start menu, type Windows PowerShell, select Windows PowerShell, then select Open.

Run from the Command Prompt

In Windows Command shell, Windows PowerShell, or Windows PowerShell ISE, to start Windows PowerShell, type: `PowerShell`.

You can also use the parameters of the `powershell.exe` program to customize the session. For more information, see [about_PowerShell_exe](#).

Run with administrative privileges

Open the **Start** menu, type **Windows PowerShell**, select **Windows PowerShell**, and then select **Run as administrator**.

How to Start Windows PowerShell ISE on Earlier Releases of Windows

Use any of the following methods to start Windows PowerShell ISE.

Run from the Start Menu

- Open the **Start** menu, type **ISE**, select **Windows PowerShell ISE**, then select **Open**.

At the Command Prompt

In Windows Command shell, Windows PowerShell, or Windows PowerShell ISE, to start Windows PowerShell, type: `PowerShell_ISE`. In Windows PowerShell, you can use the alias `ise`.

Run with administrative privileges

Select **Start**, type **ISE**, right-click **Windows PowerShell ISE**, and then click **Run as administrator**.

Starting the 32-Bit Version of Windows PowerShell

64-bit versions of Windows include a 32-bit version of Windows PowerShell, **Windows PowerShell (x86)**, in addition to the 64-bit version. The 64-bit version runs by default.

However, you might occasionally need to run **Windows PowerShell (x86)**, such as when you're using a module that requires the 32-bit version or when you're connecting

remotely to a 32-bit computer.

To start a 32-bit version of Windows PowerShell, use any of the following procedures.

- Select **Start**, type **Windows PowerShell**, select **Windows PowerShell (x86)**, then select **Open**.

Windows Management Framework

Article • 03/27/2025

Windows Management Framework (WMF) provides a consistent management interface for Windows. WMF provides a seamless way to manage various versions of Windows client and Windows Server. WMF installer packages contain updates to management functionality and are available for older versions of Windows.

Note

WMF 5.1 is the only supported version of WMF and is included in all currently supported versions of Windows. This information in this article provides a history of WMF versions.

WMF installation adds and/or updates the following features:

- Windows PowerShell
- Windows PowerShell Desired State Configuration (DSC)
- Windows PowerShell Integrated Script Environment (ISE)
- Windows Remote Management (WinRM)
- Windows Management Instrumentation (WMI)
- Windows PowerShell Web Services (Management OData IIS Extension)
- Software Inventory Logging (SIL)
- Server Manager CIM Provider

WMF Release Notes

To learn about the enhancements in Windows PowerShell and other components, see the release notes for each version of WMF:

- [WMF 5.1](#)
- [WMF 5.0](#)

WMF availability across Windows operating systems

 Expand table

OS Version	End of Support	WMF 5.1	WMF 5.0	WMF 4.0	WMF 3.0	WMF 2.0
Windows Server 2022	2031-10-14	Included				
Windows Server 2019	2029-01-09	Included				
Windows Server 2016	2027-01-11	Included				
Windows 11	2025-10-14	Included				
Windows 10	2025-10-14	Included in 1607+	Included			
Windows Server 2012 R2	<i>Out of support</i>	Yes ↗	Yes	Included		
Windows 8.1	<i>Out of support</i>	Yes ↗	Yes	Included		
Windows Server 2012	<i>Out of support</i>	Yes ↗	Yes	Yes	Included	
Windows 8	<i>Out of support</i>				Included	
Windows Server 2008 R2 SP1	<i>Out of support</i>	Yes ↗	Yes	Yes	Yes	Included
Windows 7 SP1	<i>Out of support</i>	Yes ↗	Yes	Yes	Yes	Included
Windows Server 2008 SP2	<i>Out of support</i>			Yes	Yes	
Windows Vista	<i>Out of support</i>				Yes	
Windows Server 2003	<i>Out of support</i>				Yes	
Windows XP	<i>Out of support</i>			Yes	Yes	

- **Included:** The features of the specified version of WMF were shipped in the indicated version of Windows client or Windows Server.
- **Out of support:** Microsoft no longer supports these products. You must upgrade to a supported version. For more information, see the [Microsoft Lifecycle Policy ↗](#)

page.

 **Note**

The version of WMF that shipped in an operating system is supported for the lifetime of support for that version of the operating system. The standalone installers for WMF 5.0 and older are no longer available or supported.

PowerShell Security

Learn about PowerShell's security features and best practices.

Security features

OVERVIEW

[PowerShell security features](#)

[Using App Control for Business](#)

HOW-TO GUIDE

[Preventing script injection attacks](#)

[Securing a restricted PowerShell remoting session](#)

PowerShell remoting

CONCEPT

[Running remote commands](#)

[Using WS-Management \(WSMan\) Remoting in PowerShell](#)

[Security Considerations for PowerShell Remoting using WinRM](#)

[PowerShell Remoting FAQ](#)

HOW-TO GUIDE

[Making the second hop in PowerShell Remoting](#)

[PowerShell remoting over SSH](#)

Just Enough Administration (JEA)

CONCEPT

[Overview](#)

[Prerequisites](#)

[JEA Role Capabilities](#)

[Session configurations](#)

[Security considerations](#)

HOW-TO GUIDE

[Registering JEA Configurations](#)

[Using JEA](#)

[Auditing and Reporting on JEA](#)

Using App Control

OVERVIEW

[Using App Control for Business](#)

[How App Control works with PowerShell](#)

HOW-TO GUIDE

[How to use App Control to secure PowerShell](#)

Managing secrets

CONCEPT

[Overview of the SecretManagement and SecretStore modules](#)

[Understanding the security features of SecretManagement and SecretStore](#)

HOW-TO GUIDE

[Managing a SecretStore vault](#)

[Use the SecretStore in automation](#)

[Use Azure Key Vault in automation](#)

REFERENCE

[Microsoft.PowerShell.SecretManagement module](#)

[Microsoft.PowerShell.SecretStore module](#)

PowerShell security features

Article • 05/22/2025

PowerShell has several features designed to improve the security of your scripting environment.

Execution policy

PowerShell's execution policy is a safety feature that controls the conditions under which PowerShell loads configuration files and runs scripts. This feature helps prevent the execution of malicious scripts. You can use a Group Policy setting to set execution policies for computers and users. Execution policies only apply to the Windows platform.

For more information, see [about_Execution_Policies](#).

Use of the `SecureString` class

PowerShell has several cmdlets that support the use of the `System.Security.SecureString` class. And, as with any .NET class, you can use `SecureString` in your own scripts. However, Microsoft doesn't recommend using `SecureString` for new development. Microsoft recommends that you avoid using passwords and rely on other means to authenticate, such as certificates or Windows authentication.

PowerShell continues to support the `SecureString` class for backward compatibility. Using a `SecureString` is still more secure than using a plain text string. PowerShell still relies on the `SecureString` type to avoid accidentally exposing the contents to the console or in logs. Use `SecureString` carefully, because it can be easily converted to a plain text string. For a full discussion about using `SecureString`, see the [System.Security.SecureString class](#) documentation.

Module and script block logging

Module Logging allows you to enable logging for selected PowerShell modules. This setting is effective in all sessions on the computer. PowerShell records pipeline execution events for the specified modules in the Windows PowerShell event log.

Script Block Logging enables logging for the processing of commands, script blocks, functions, and scripts - whether invoked interactively, or through automation. PowerShell logs this information to the **Microsoft-Windows-PowerShell/Operational** event log.

For more information, see the following articles:

- [about_Group_Policy_Settings](#)
- [about_Logging_Windows](#)
- [about_Logging_Non-Windows](#)

AMSI Support

The Windows Antimalware Scan Interface (AMSI) is an API that allows applications to pass actions to an antimalware scanner, such as Windows Defender, to scan for malicious payloads. Beginning with PowerShell 5.1, PowerShell running on Windows 10 (and higher) passes all script blocks to AMSI.

PowerShell 7.3 extends the data it sends to AMSI for inspection. It now includes all .NET method invocations.

For more information about AMSI, see [How AMSI helps](#).

Constrained language mode

`ConstrainedLanguage` mode protects your system by limiting the cmdlets and .NET types allowed in a PowerShell session. For a full description, see [about_Language_Modes](#).

Application Control

Windows 10 includes two technologies, [App Control for Business](#) and [AppLocker](#) that you can use to control applications. PowerShell detects if a system wide application control policy is being enforced. The policy applies certain behaviors when running script blocks, script files, or loading module files to prevent arbitrary code execution on the system.

App Control for Business is designed as a security feature under the servicing criteria defined by the Microsoft Security Response Center (MSRC). App Control is the preferred application control system for Windows.

For more information about how PowerShell supports AppLocker and App Control, see [Use App Control to secure PowerShell](#).

Software Bill of Materials (SBOM)

Beginning with PowerShell 7.2, all install packages contain a Software Bill of Materials (SBOM). The PowerShell team also produces SBOMs for modules that they own but ship independently from PowerShell.

You can find SBOM files in the following locations:

- In PowerShell, find the SBOM at `$PSHOME/_manifest/spdx_2.2/manifest.spdx.json`.
- For modules, find the SBOM in the module's folder under
`_manifest/spdx_2.2/manifest.spdx.json`.

The creation and publishing of the SBOM is the first step to modernize Federal Government cybersecurity and enhance software supply chain security. For more information about this initiative, see the blog post [Generating SBOMs with SPDX at Microsoft](#).

Security Servicing Criteria

PowerShell follows the [Microsoft Security Servicing Criteria for Windows](#). Only security features meet the criteria for servicing.

Security features

- System Lockdown with App Control for Business
- Constrained language mode with App Control for Business

Defense in depth features

- System Lockdown with AppLocker
- Constrained language mode with AppLocker
- Execution Policy

Use App Control to secure PowerShell

Article • 10/21/2024

Windows 10 includes two technologies, [App Control for Business](#) and [AppLocker](#), that you can use to control applications. They allow you to create a lockdown experience to help secure your PowerShell environment.

AppLocker builds on the application control features of Software Restriction Policies. AppLocker allows you to create rules to allow or deny apps for specific users or groups. You identify the apps based on unique properties of the files.

Application Control for Business, introduced in Windows 10 as Windows Defender Application Control (WDAC), allows you to control which drivers and applications are allowed to run on Windows.

Lockdown policy detection

PowerShell detects both AppLocker and App Control for Business system wide policies. AppLocker doesn't have way to query the policy enforcement status. To detect if a system wide application control policy is being enforced by AppLocker, PowerShell creates two temporary files and tests if they can be executed. The filenames use the following name format:

- `$Env:TEMP/_PSAppLockerTest_\<random-8dot3-name>.ps1`
- `$Env:TEMP/_PSAppLockerTest_\<random-8dot3-name>.psm1`

App Control for Business is the preferred application control system for Windows. App Control provides APIs that allow you to discover the policy configuration. App Control is designed as a security feature under the servicing criteria defined by the Microsoft Security Response Center (MSRC). For more information, see [Application Controls for Windows](#) and [App Control and AppLocker feature availability](#).

ⓘ Note

When [choosing between App Control or AppLocker](#), we recommend that you implement application control using App Control for Business rather than AppLocker. Microsoft is no longer investing in AppLocker. Although AppLocker may continue to receive security fixes, it won't receive feature enhancements.

App Control policy enforcement

When PowerShell runs under an App Control policy, its behavior changes based on the defined security policy. Under an App Control policy, PowerShell runs trusted scripts and modules allowed by the policy in `FullLanguage` mode. All other scripts and script blocks are untrusted and run in `ConstrainedLanguage` mode. PowerShell throws errors when the untrusted scripts attempt to perform actions that aren't allowed in `ConstrainedLanguage` mode. It can be difficult to know why a script failed to run correctly in `ConstrainedLanguage` mode.

App Control policy auditing

PowerShell 7.4 added a new feature to support App Control policies in **Audit** mode. In audit mode, PowerShell runs the untrusted scripts in `ConstrainedLanguage` mode without errors, but logs messages to the event log instead. The log messages describe what restrictions would apply if the policy were in **Enforce** mode.

History of changes

Windows PowerShell 5.1 was the first version of PowerShell to support App Control. The security features of App Control and AppLocker improve with each new release of PowerShell. The following sections describe how this support changed in each version of PowerShell. The changes are cumulative, so the features described in the later versions include those from earlier versions.

Changes in PowerShell 7.4

On Windows, when PowerShell runs under an App Control policy, its behavior changes based on the defined security policy. Under an App Control policy, PowerShell runs trusted scripts and modules allowed by the policy in `FullLanguage` mode. All other scripts and script blocks are untrusted and run in `ConstrainedLanguage` mode.

PowerShell throws errors when the untrusted scripts attempt to perform disallowed actions. It's difficult to know why a script fails to run correctly in `ConstrainedLanguage` mode.

PowerShell 7.4 now supports App Control policies in **Audit** mode. In audit mode, PowerShell runs the untrusted scripts in `ConstrainedLanguage` mode but logs messages to the event log instead of throwing errors. The log messages describe what restrictions would apply if the policy were in **Enforce** mode.

Changes in PowerShell 7.3

- PowerShell 7.3 now supports the ability to block or allow PowerShell script files via the App Control API.

Changes in PowerShell 7.2

- There was a corner-case scenario in AppLocker where you only have **Deny** rules and constrained mode isn't used to enforce the policy that allows you to bypass the execution policy. Beginning in PowerShell 7.2, a change was made to ensure AppLocker rules take precedence over a `Set-ExecutionPolicy -ExecutionPolicy Bypass` command.
- PowerShell 7.2 now disallows the use of the `Add-Type` cmdlet in a `NoLanguage` mode PowerShell session on a locked down machine.
- PowerShell 7.2 now disallows scripts from using COM objects in AppLocker system lockdown conditions. Cmdlets that use COM or DCOM internally aren't affected.

Further reading

- For more information about how App Control works and what restrictions it enforces, see [How App Control works with PowerShell](#).
- For more information about securing PowerShell with App Control, see [How to use App Control](#).

How App Control works with PowerShell

Article • 10/21/2024

This article explains how **App Control for Business** secures PowerShell and the restrictions it imposes. The secure behavior of PowerShell varies based on the version of Windows and PowerShell you're using.

How PowerShell detects a system lockdown policy

PowerShell detects both **AppLocker** and **App Control for Business** system wide policies. AppLocker is deprecated. App Control is the preferred application control system for Windows.

Legacy App Control policy enforcement detection

PowerShell uses the legacy App Control `W1dpGetLockdownPolicy` API to discover two things:

- System wide policy enforcement: `None`, `Audit`, `Enforce`
- Individual file policy: `None`, `Audit` (allowed by policy), `Enforce` (not allowed by policy)

All versions of PowerShell (v5.1 - v7.x) support this App Control policy detection.

Latest App Control policy enforcement detection

App Control introduced new APIs in recent versions of Windows. Beginning with version 7.3, PowerShell uses the new `W1dpCanExecuteFile` API to decide how a file should be handled. Windows PowerShell 5.1 doesn't support this new API. The new API takes precedence over the legacy API for individual files. However, PowerShell continues to use the legacy API to get the system wide policy configuration. If the new API isn't available, PowerShell falls back to the old API behavior.

The new API provides the following information for each file:

- `WLDP_CAN_EXECUTE_ALLOWED`
- `WLDP_CAN_EXECUTE_BLOCKED`
- `WLDP_CAN_EXECUTE_REQUIRE_SANDBOX`

PowerShell behavior under lockdown policy

PowerShell can run in both interactive and non-interactive modes.

- In interactive mode, PowerShell is a command-line application that takes users command-line input as commands or scripts to run. Results are displayed back to the user.
- In non-interactive mode, PowerShell loads modules and runs script files without user input. Result data streams are either ignored or redirected to a file.

Interactive mode running under policy enforcement

PowerShell runs commands in `ConstrainedLanguage` mode. This mode prevents interactive users from running certain commands or executing arbitrary code. For more information about the restrictions in this mode, see the [PowerShell restrictions under lockdown policy](#) section of this article.

Noninteractive mode running under policy enforcement

When PowerShell runs a script or loads a module, it uses the App Control API to get the policy enforcement for the file.

PowerShell version 7.3 or higher uses the `W1dpCanExecuteFile` API if available. This API returns one of the following results:

- `WLDP_CAN_EXECUTE_ALLOWED`: The file is approved by policy and is used in `FullLanguage` mode with a few restrictions.
- `WLDP_CAN_EXECUTE_BLOCKED`: The file isn't approved by policy. PowerShell throws an error when the file is run or loaded.
- `WLDP_CAN_EXECUTE_REQUIRE_SANDBOX`: The file isn't approved by the policy but it can still be run or loaded in `ConstrainedLanguage` mode.

In Windows PowerShell 5.1 or if `W1dpCanExecuteFile` API isn't available, PowerShell's per file behavior is:

- `None`: The file is run loaded in `FullLanguage` mode with a few restrictions.
- `Audit`: The file is run or loaded in `FullLanguage` mode with no restrictions. In PowerShell 7.4 or higher, the policy logs restriction information to the Windows event logs.
- `Enforce`: The file is run or loaded in `ConstrainedLanguage` mode.

PowerShell restrictions under lockdown policy

When PowerShell detects the system is under a App Control lockdown policy, it applies restrictions even if the script is trusted and running in `FullLanguage` mode. These restrictions prevent known behaviors of PowerShell that could result in arbitrary code execution on a locked-down system. The lockdown policy enforces the following restrictions:

- Module dot-sourcing with wildcard function export restriction

Any module that uses script dot-sourcing and exports functions using wildcard names results in an error. Blocking wildcard exports prevents script injection from a malicious user who can plant an untrusted script that gets dot-sourced into a trusted module. The malicious script could then gain access to the trusted module's private functions.

Security recommendation: Never use script dot-sourcing in a module and always export module functions with explicit names (no wildcard characters).

- Nested module with wildcard function export restriction

If a parent module exports functions using function name wildcard characters, PowerShell removes any function name in a nested module from the function export list. Blocking wildcard exports from nested modules prevents accidental exporting of dangerous nested functions through wildcard name matching.

Security recommendation: Always export module functions with explicit names (no wildcard characters).

- Interactive shell parameter type conversion

When the system is locked down, interactive PowerShell sessions run in `ConstrainedLanguage` mode to prevent arbitrary code execution. Trusted modules loaded into the session run in `FullLanguage` mode. If a trusted module cmdlet uses complex types for its parameters, type conversion during parameter binding can fail if the conversion isn't allowed across trust boundaries. The failure occurs when PowerShell tries to convert a value by running a type constructor. Type constructors aren't allowed to run in `ConstrainedLanguage` mode.

In this example, parameter binding type conversion is normally allowed, but fails when run in `ConstrainedLanguage` mode. The `ConnectionPort` type constructor isn't allowed:

```
PowerShell

PS> Create-ConnectionOnPort -Connection 22
Create-ConnectionOnPort: Cannot bind parameter 'Connection'. Cannot convert
the "22"
value of type "System.Int32" to type "ConnectionPort".
```

- `Enter-PSTokenHost` cmdlet disallowed

The `Enter-PSTokenHost` cmdlet is disabled and throws an error if used. This command is used for attach-and-debug sessions. It allows you to connect to any other PowerShell

session on the local machine. The cmdlet is disabled to prevent information disclosure and arbitrary code execution.

PowerShell restrictions under constrained language mode

Script or function that isn't approved by the App Control policy is untrusted. When you run an untrusted command, PowerShell either blocks the command from running (new behavior) or runs the command in `ConstrainedLanguage` mode. The following restrictions apply to

`ConstrainedLanguage` mode:

- `Add-Type` cmdlet disallowed

Blocking `Add-Type` prevents the execution of arbitrary .NET code.

- `Import-LocalizedData` cmdlet restricted

Blocking the `SupportedCommand` parameter of `Import-LocalizedData` prevents the execution of arbitrary code.

- `Invoke-Expression` cmdlet restricted

All script blocks passed to the `Invoke-Expression` cmdlet are run in `ConstrainedLanguage` mode to prevent arbitrary code execution.

- `New-Object` cmdlet restricted

The `New-Object` cmdlet is restricted to use only allowed .NET and COM types, to prevent access to untrusted types.

- `ForEach-Object` cmdlet restriction

Type method invocation for variables passed to the `ForEach-Object` is disallowed for any .NET type not in the approved list. In general, `ConstrainedLanguage` mode disallows any object method invocation except for approved .NET types to prevent access to untrusted .NET types.

- `Export-ModuleMember` cmdlet restriction

Using `Export-ModuleMember` cmdlet to export functions in a nested module script file where the child module isn't trusted and the parent module is trusted, results in an error. Blocking this scenario prevents a malicious untrusted module from exporting dangerous functions from a trusted module.

- `New-Module` cmdlet restriction

When you run `New-Module` in a trusted script, any script block provided by the `ScriptBlock` parameter is marked to run in `ConstrainedLanguage` mode to prevent the injection of arbitrary code into a trusted execution context.

- `Configuration` keyword not allowed

The `Configuration` language keyword isn't allowed in `ConstrainedLanguage` mode to prevent code injection attacks.

- `class` keyword not allowed

The `class` language keyword isn't allowed in `ConstrainedLanguage` mode to prevent the injection of arbitrary code.

- Script Block processing scope restrictions

Child script blocks aren't allowed to run in parent script block scopes if the script blocks have different trust levels. For example, you create a child relationship when you dot-source one script into another. Blocking this scenario prevents an untrusted script from getting access to dangerous functions in the trusted script scope.

- Prevent command discovery of untrusted script functions

PowerShell command discovery doesn't return functions from an untrusted source, such as an untrusted script or module, to a trusted function. Blocking discovery of untrusted commands prevents code injection through command planting.

- Hashtable to object conversion not allowed

`ConstrainedLanguage` mode blocks hashtable to object conversions in the `Data` sections of PowerShell data (`.psd1`) files to prevent potential code injection attacks.

- Automatic type conversion restricted

`ConstrainedLanguage` mode blocks automatic type conversion except for a small set of approved safe types to prevent potential code injection attacks.

- Implicit module function export restriction

If a module doesn't explicitly export functions, PowerShell implicitly exports all defined module functions automatically as a convenience feature. In `ConstrainedLanguage` mode, implicit exports no longer occur when a module is loaded across trust boundaries.

Blocking implicit exports prevents unintended exposure of dangerous module functions not meant for public use.

- Script files can't be imported as modules

PowerShell allows you to import script files (`.ps1`) as a module. All defined functions become publicly accessible. `ConstrainedLanguage` mode blocks importation of script file to prevent unintended exposure of dangerous script functions.

- Setting variables `AllScope` restriction

`ConstrainedLanguage` mode disables the ability to set `AllScope` on variables. Limiting the scope of variables prevents the variables from interfering with the session state of trusted commands.

- Type method invocation not allowed

`ConstrainedLanguage` mode doesn't allow method invocation on unapproved types.

Blocking methods on unapproved types prevents invocation of .NET type methods that might be dangerous or allow code injection.

- Type property setters not allowed

`ConstrainedLanguage` mode restricts invocation of property setters on unapproved types.

Blocking property setters on unapproved types prevents code injection attacks.

- Type creation not allowed

`ConstrainedLanguage` mode blocks type creation on unapproved types to block untrusted constructors that could allow code injection.

- Module scope operator not allowed

`ConstrainedLanguage` mode doesn't allow the use of the module scope operator. For example: `& (Get-Module MyModule) MyFunction`. Blocking the module scope operator prevents access to module private functions and variables.

Further reading

- For more information about PowerShell language modes, see [about_Language_Modes](#).
- For information about how to configure and use App Control, see [How to use App Control for PowerShell](#).

How to use App Control to secure PowerShell

Article • 10/21/2024

This article describes how to set up a **App Control for Business** policy. You can configure the policy to enforce or audit the policy's rule. In audit mode, PowerShell behavior doesn't change but it logs Event ID 16387 messages to the `PowerShellCore/Analytic` event log. In enforcement mode, PowerShell applies the policy's restrictions.

This article assumes you're using a test machine so that you can test PowerShell behavior under a machine wide App Control policy before you deploy the policy in your environment.

Create an App Control policy

An App Control policy is described in an XML file, which contains information about policy options, files allowed, and signing certificates recognized by the policy. When the policy is applied, only approved files are allowed to load and run. PowerShell either blocks unapproved script files from running or runs them in `ConstrainedLanguage` mode, depending on policy options.

You create and manipulate App Control policy using the **ConfigCI** module, which is available on all supported Windows versions. This Windows PowerShell module can be used in Windows PowerShell 5.1 or in PowerShell 7 through the **Windows Compatibility** layer. It's easier to use this module in Windows PowerShell. The policy you create can be applied to any version of PowerShell.

Steps to create an App Control policy

For testing, you just need to create a default policy and a self signed code signing certificate.

1. Create a default policy

```
PowerShell  
New-CIPolicy -Level PcaCertificate -FilePath .\SystemCIPolicy.xml -UserPEs
```

This command creates a default policy file called `SystemCIPolicy.xml` that allows all Microsoft code-signed files to run.

 Note

Running this command can take up to two hours because it must scan the entire test machine.

2. Disable Audit Mode in default policy

A new policy is always created in `Audit` mode. To test policy enforcement, you need to disable Audit mode when you apply the policy. Edit the `SystemCIPolicy.xml` file using a text editor like `notepad.exe` or Visual Studio Code (VS Code). Comment out the `Audit mode` option.

XML

```
<!--
<Rule>
  <Option>Enabled:Audit Mode</Option>
</Rule>
-->
```

3. Create a self-signed code signing certificate

You need a code signing certificate to sign any test binaries or script files that you want to run on your test machine. The `New-SelfSignedCertificate` is provided by the **PKI** module. For best results, you should run this command in Windows PowerShell 5.1.

PowerShell

```
$newSelfSignedCertificateSplat = @{
    DnsName = $Env:COMPUTERNAME
    CertStoreLocation = "Cert:\CurrentUser\My\" 
    Type = 'CodeSigningCert'
}
$cert = New-SelfSignedCertificate @newSelfSignedCertificateSplat
Export-Certificate -Cert $cert -FilePath C:\certs\signing.cer
Import-Certificate -FilePath C:\certs\signing.cer -CertStoreLocation
"Cert:\CurrentUser\Root\
$cert = Get-ChildItem Cert:\CurrentUser\My\ -CodeSigningCert

dir C:\bin\PowerShell\pwsh.exe | Set-AuthenticodeSignature -Certificate $cert
```

4. Add the code signing certificate to the policy

Use the following command to add the new code signing certificate to the policy.

PowerShell

```
Add-SignerRule -FilePath .\SystemCIPolicy.xml -CertificatePath
```

```
C:\certs\signing.cer -User
```

5. Convert the XML policy file to a policy enforcement binary file

Finally, you need to convert the XML file to a binary file used by App Control to apply a policy.

PowerShell

```
ConvertFrom-CIPolicy -XmlFilePath .\SystemCIPolicy.xml -BinaryFilePath  
.\\SIPolicy.p7b
```

6. Apply the App Control policy

To apply the policy to your test machine, copy the `SIPolicy.p7b` file to the required system location, `C:\Windows\System32\CodeIntegrity`.

ⓘ Note

Some policies definition must be copied to a subfolder such as

`C:\Windows\System32\CodeIntegrity\CiPolicies`. For more information, see [App Control Admin Tips & Known Issues](#).

7. Disable the App Control policy

To disable the policy, rename the `SIPolicy.p7b` file. If you need to do more testing, you can change the name back to reenable the policy.

PowerShell

```
Rename-Item -Path .\\SIPolicy.p7b -NewName .\\SIPolicy.p7b.off
```

Test using App Control policy auditing

PowerShell 7.4 added a new feature to support App Control policies in **Audit** mode. In audit mode, PowerShell runs the untrusted scripts in `ConstrainedLanguage` mode without errors, but logs messages to the event log instead. The log messages describe what restrictions would apply if the policy were in **Enforce** mode.

Viewing audit events

PowerShell logs audit events to the **PowerShellCore/Analytic** event log. The log isn't enabled by default. To enable the log, open the **Windows Event Viewer**, right-click on the **PowerShellCore/Analytic** log and select **Enable Log**.

Alternatively, you can run the following command from an elevated PowerShell session.

```
PowerShell  
  
wevtutil.exe sl PowerShellCore/Analytic /enabled:true /quiet
```

You can view the events in the Windows Event Viewer or use the `Get-WinEvent` cmdlet to retrieve the events.

```
PowerShell  
  
Get-WinEvent -LogName PowerShellCore/Analytic -Oldest |  
    Where-Object Id -EQ 16387 | Format-List
```

```
Output  
  
TimeCreated : 4/19/2023 10:11:07 AM  
ProviderName : PowerShellCore  
Id          : 16387  
Message      : App Control Audit.  
  
    Title: Method or Property Invocation  
    Message: Method or Property 'WriteLine' on type 'System.Console' invocation  
will not  
        be allowed in ConstrainedLanguage mode.  
        At C:\scripts\Test1.ps1:3 char:1  
        + [System.Console]::WriteLine("pwd!")  
        + ~~~~~  
FullyQualifiedId: MethodOrPropertyInvocationNotAllowed
```

The event message includes the script position where the restriction would be applied. This information helps you understand where you need to change your script so that it runs under the App Control policy.

ⓘ Important

Once you have reviewed the audit events, you should disable the Analytic log. Analytic logs grow quickly and consume large amounts of disk space.

Viewing audit events in the PowerShell debugger

If you set the `$DebugPreference` variable to `Break` for an interactive PowerShell session, PowerShell breaks into the command-line script debugger at the current location in the script where the audit event occurred. The breakpoint allows you to debug your code and inspect the current state of the script in real time.

Preventing script injection attacks

Article • 04/01/2024

PowerShell scripts, like other programming languages, can be vulnerable to injection attacks. An injection attack occurs when a user provides input to a vulnerable function that includes extra commands. The vulnerable function runs the extra commands, which can be a serious security vulnerability. For example, a malicious user could abuse the vulnerable function to run arbitrary code on a remote computer, possibly compromising that computer and gaining access to other machines on the network.

Once you are aware of the issue, there are several ways to protect against injection attacks.

Example of vulnerable code

PowerShell code injection vulnerabilities involve user input that contains script code. The user input is added to vulnerable script where it's parsed and run by PowerShell.

```
PowerShell

function Get-ProcessById
{
    param ($ProcId)

    Invoke-Expression -Command "Get-Process -Id $ProcId"
}
```

The `Get-ProcessById` function looks up a local process by its Id value. It takes a `$ProcId` parameter argument of any type. The `$ProcId` is then converted to a string and inserted into another script that's parsed and run using the `Invoke-Expression` cmdlet. This function works fine when a valid process Id integer is passed in.

```
PowerShell

Get-ProcessById $PID

  NPM(K)   PM(M)   WS(M)   CPU(s)      Id  SI ProcessName
  -----  -----  -----  -----  --  --  -----
  97      50.09   132.72   1.20     12528  3  pwsh
```

However, the `$ProcId` parameter doesn't specify a type. It accepts any arbitrary string value that can include other commands.

```
PowerShell
```

```
Get-ProcessById "$PID; Write-Host 'pwnd!'"
```

In this example, the function correctly retrieved the process identified by `$PID`, but also ran the injected script `Write-Host 'pwnd!'`.

```
Output
```

NPM(K)	PM(M)	WS(M)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	--	--	-----
92	45.66	122.52	1.06	21736	3	pwsh
pwnd!						

Ways to guard against injection attacks

There are several ways to guard against an injection attack.

Use typed input

You can specify a type for the `$ProcId` argument.

```
PowerShell
```

```
function Get-ProcessById
{
    param ([int] $ProcId)

    Invoke-Expression -Command "Get-Process -Id $ProcId"
}
Get-ProcessById "$PID; Write-Host 'pwnd!'"
```

```
Output
```

```
Get-ProcessById:
Line |
7 | Get-ProcessById "$PID; Write-Host 'pwnd!'" |
| ~~~~~
| Cannot process argument transformation on parameter 'ProcId'. Cannot
convert value
"8064; Write-Host 'pwnd!'" to type "System.Int32". Error: "The input string
'8064; Write-Host 'pwnd!'
was not in a correct format."
```

Here, the `$ProcId` input parameter is restricted to an integer type, so an error occurs when a string is passed in that can't be converted to an integer.

Don't use `Invoke-Expression`

Instead of using `Invoke-Expression`, directly call `Get-Process`, and let PowerShell's parameter binder validate the input.

```
PowerShell

function Get-ProcessById
{
    param ($ProcId)

    Get-Process -Id $ProcId
}
Get-ProcessById "$PID; Write-Host 'pwnd!'"
```

Output

```
Get-Process:
Line |
 5 |     Get-Process -Id $ProcId
   |     ~~~~~
   | Cannot bind parameter 'Id'. Cannot convert value "8064; Write-Host
'pwnd!'" to type
[System.Int32]. Error: "The input string '8064; Write-Host 'pwnd!' was not
in a correct
format."
```

As a best practice, you should avoid using `Invoke-Expression`, especially when handling user input. `Invoke-Expression` is dangerous because it parses and runs whatever string content you provide, making it vulnerable to injection attacks. It's better to rely on PowerShell parameter binding.

Wrap strings in single quotes

However, there are times when using `Invoke-Expression` is unavoidable and you also need to handle user string input. You can safely handle user input using single quotes around each string input variable. The single quote ensures that PowerShell's parser treats the user input as a single string literal.

PowerShell

```

function Get-ProcessById
{
    param ($ProcId)

    Invoke-Expression -Command "Get-Process -Id '$ProcId'"

}

Get-ProcessById "$PID; Write-Host 'pwnd!'"
```

Output

```

Get-Process: Cannot bind parameter 'Id'. Cannot convert value "8064; Write-
Host " to type
"System.Int32". Error: "The input string '8064; Write-Host' was not in a
correct format."
```

However, this version of the function isn't yet completely safe from injection attacks. A malicious user can still use single quotes in their input to inject code.

PowerShell

```
Get-ProcessById "$PID'; Write-Host 'pwnd!';'"
```

This example uses single quotes in the user input to force the function to run three separate statements, one of which is arbitrary code injected by the user.

Output

NPM(K)	PM(M)	WS(M)	CPU(s)	Id	SI	ProcessName
97	46.08	183.10	1.08	2524	3	pwsh
pwnd!						

Use the `EscapeSingleQuotedStringContent()` method

To protect against the user inserting their own single quote characters to exploit the function you must use the `EscapeSingleQuotedStringContent()` API. This is a static public method of the PowerShell

`System.Management.Automation.Language.CodeGeneration` class. This method makes the user input safe by escaping any single quotes included in the user input.

PowerShell

```
function Get-ProcessById
{
    param ($ProcId)

    $ProcIdClean = [System.Management.Automation.Language.CodeGeneration]::
        EscapeSingleQuotedStringContent("$ProcId")
    Invoke-Expression -Command "Get-Process -Id '$ProcIdClean'"
}

Get-ProcessById "$PID"; Write-Host 'pwnd!'; ''
```

Output

```
Get-Process: Cannot bind parameter 'Id'. Cannot convert value "8064"; Write-
Host 'pwnd!'; '' to type
"System.Int32". Error: "The input string '8064'; Write-Host 'pnd!'; '' was
not in a correct format."
```

For more information, see [EscapeSingleQuotedStringContent\(\)](#).

Detecting vulnerable code with Injection Hunter

Injection Hunter is a module written by Lee Holmes that contains PowerShell Script Analyzer rules for detecting code injection vulnerabilities. Use one of the following commands to install the module from the PowerShell Gallery:

PowerShell

```
# Use PowerShellGet v2.x
Install-Module InjectionHunter

# Use PowerShellGet v3.x
Install-PSResource InjectionHunter
```

You can use this to automate security analysis during builds, continuous integration processes, deployments, and other scenarios.

PowerShell

```
$RulePath = (Get-Module -List InjectionHunter).Path
Invoke-ScriptAnalyzer -CustomRulePath $RulePath -Path .\Invoke-Dangerous.ps1
```

Output

RuleName	Severity	ScriptName	Line	Message
-----	-----	-----	-----	-----
InjectionRisk.InvokeExpression	Warning	Invoke-Dan	3	Possible
script injection risk via the		gerous.ps1		
Expression cmdlet. Untrusted input can cause				Invoke-
PowerShell expressions to be run.				arbitrary
may be used directly for dynamic parameter				Variables
splatting can be used for dynamic				arguments,
names, and the invocation operator can be				parameter
dynamic command names. If content escaping				used for
needed, PowerShell has several valid quote				is truly
characters, so [System.Management.Automation.Languag				
e.CodeGeneration]::Escape* should be used.				

For more information, see [PSScriptAnalyzer](#).

Related links

- [Lee Holmes' blog post about Injection Hunter ↗](#)
- [Injection Hunter ↗](#)

Just Enough Administration

Article • 04/01/2024

Just Enough Administration (JEA) is a security technology that enables delegated administration for anything managed by PowerShell. With JEA, you can:

- **Reduce the number of administrators on your machines** using virtual accounts or group-managed service accounts to perform privileged actions on behalf of regular users.
- **Limit what users can do** by specifying which cmdlets, functions, and external commands they can run.
- **Better understand what your users are doing** with transcripts and logs that show you exactly which commands a user executed during their session.

Why is JEA important?

Highly privileged accounts used to administer your servers pose a serious security risk. Should an attacker compromise one of these accounts, they could launch [lateral attacks](#) across your organization. Each compromised account gives an attacker access to even more accounts and resources, and puts them one step closer to stealing company secrets, launching a denial-of-service attack, and more.

It's not always easy to remove administrative privileges, either. Consider the common scenario where the DNS role is installed on the same machine as your Active Directory Domain Controller. Your DNS administrators require local administrator privileges to fix issues with the DNS server. But to do so, you must make them members of the highly privileged **Domain Admins** security group. This approach effectively gives DNS Administrators control over your whole domain and access to all resources on that machine.

JEA addresses this problem through the principle of **Least Privilege**. With JEA, you can configure a management endpoint for DNS administrators that gives them access only to the PowerShell commands they need to get their job done. This means you can provide the appropriate access to repair a poisoned DNS cache or restart the DNS server without unintentionally giving them rights to Active Directory, or to browse the file system, or run potentially dangerous scripts. Better yet, when the JEA session is configured to use temporary privileged virtual accounts, your DNS administrators can connect to the server using **non-admin** credentials and still run commands that typically require admin privileges. JEA enables you to remove users from widely privileged local/domain administrator roles and carefully control what they can do on each machine.

Next steps

To learn more about the requirements to use JEA, see the [Prerequisites](#) article.

Samples and DSC resource

Sample JEA configurations and the JEA DSC resource can be found in the [JEA GitHub repository](#).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

JEA Prerequisites

Article • 04/01/2024

Just Enough Administration is a feature included in PowerShell 5.0 and higher. This article describes the prerequisites that must be satisfied to start using JEA.

Check which version of PowerShell is installed

To check which version of PowerShell is installed on your system, check the

`$PSVersionTable` variable in a Windows PowerShell prompt.

```
PowerShell
$PSVersionTable.PSVersion
Output
Major   Minor   Build   Revision
-----  -----  -----  -----
5       1        14393  1000
```

JEA is available with PowerShell 5.0 and higher. For full functionality, it's recommended that you install the latest version of PowerShell available for your system. The following table describes JEA's availability on Windows Server:

[] Expand table

Server Operating System	JEA Availability
Windows Server 2016+	Preinstalled
Windows Server 2012 R2	Full functionality with WMF 5.1
Windows Server 2012	Full functionality with WMF 5.1
Windows Server 2008 R2	Reduced functionality ¹ with WMF 5.1

You can also use JEA on your home or work computer:

[] Expand table

Client Operating System	JEA Availability
Windows 10 1607+	Preinstalled
Windows 10 1603, 1511	Preinstalled, with reduced functionality ²
Windows 10 1507	Not available
Windows 8, 8.1	Full functionality with WMF 5.1
Windows 7	Reduced functionality ¹ with WMF 5.1

- ¹ JEA can't be configured to use group-managed service accounts on Windows Server 2008 R2 or Windows 7. Virtual accounts and other JEA features *are* supported.
- ² The following JEA features aren't supported on Windows 10 versions 1511 and 1603:
 - Running as a group-managed service account
 - Conditional access rules in session configurations
 - The user drive
 - Granting access to local user accounts

To get support for these features, update Windows to version 1607 (Anniversary Update) or higher.

Install Windows Management Framework

If you're running an older version of PowerShell, you may need to update your system with the latest Windows Management Framework (WMF) update. For more information, see the [WMF documentation](#).

It's recommended that you test your workload's compatibility with WMF before upgrading all of your servers.

Windows 10 users should install the latest feature updates to obtain the current version of Windows PowerShell.

Enable PowerShell Remoting

PowerShell Remoting provides the foundation on which JEA is built. It's necessary to ensure PowerShell Remoting is enabled and properly secured before you can use JEA. For more information, see [WinRM Security](#).

PowerShell Remoting is enabled by default on Windows Server 2012 and higher. You can enable PowerShell Remoting by running the following command in an elevated PowerShell window.

```
PowerShell
Enable-PSRemoting
```

Enable PowerShell module and script block logging (optional)

The following steps enable logging for all PowerShell actions on your system. PowerShell Module Logging isn't required for JEA, however it's recommended you turn on logging to ensure the commands users run are logged in a central location.

You can configure the PowerShell Module Logging policy using Group Policy.

1. Open the Local Group Policy Editor on a workstation or a Group Policy Object in the Group Policy Management Console on an Active Directory Domain Controller
2. Navigate to **Computer Configuration\Administrative Templates\Windows Components\Windows PowerShell**
3. Double-click on **Turn on Module Logging**
4. Click **Enabled**
5. In the Options section, click on **Show** next to **Module Names**
6. Type `*` in the pop-up window to log commands from all modules.
7. Click **OK** to set the policy
8. Double-click on **Turn on PowerShell Script Block Logging**
9. Click **Enabled**
10. Click **OK** to set the policy
11. (On domain-joined machines only) Run `gpupdate` or wait for Group Policy to process the updated policy and apply the settings

You can also enable system-wide PowerShell transcription through Group Policy.

Next steps

- [Create a role capability file](#)
- [Create a session configuration file](#)

See also

- WinRM Security
- PowerShell ❤ the Blue Team ↗

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

JEA Role Capabilities

Article • 04/01/2024

When creating a JEA endpoint, you need to define one or more role capabilities that describe what someone can do in a JEA session. A role capability is a PowerShell data file with the `.psrc` extension that lists all the cmdlets, functions, providers, and external programs that are made available to connecting users.

Determine which commands to allow

The first step in creating a role capability file is to consider what the users need access to. The requirements gathering process can take a while, but it's important. Giving users access to too few cmdlets and functions can prevent them from getting their job done. Allowing access to too many cmdlets and functions can allow users to do more than you intended and weaken your security stance.

How you go about this process depends on your organization and goals. The following tips can help ensure you're on the right path.

1. **Identify** the commands users are using to get their jobs done. This may involve surveying IT staff, checking automation scripts, or analyzing PowerShell session transcripts and logs.
2. **Update** use of command-line tools to PowerShell equivalents, where possible, for the best auditing and JEA customization experience. External programs can't be constrained as granularly as native PowerShell cmdlets and functions in JEA.
3. **Restrict** the scope of the cmdlets to only allow specific parameters or parameter values. This is especially important if users should manage only part of a system.
4. **Create** custom functions to replace complex commands or commands that are difficult to constrain in JEA. A simple function that wraps a complex command or applies additional validation logic can offer additional control for admins and end-user simplicity.
5. **Test** the scoped list of allowable commands with your users or automation services, and adjust as necessary.

Examples of potentially dangerous commands

Careful selection of commands is important to ensure the JEA endpoint doesn't allow the user to elevate their permissions.

 **Important**

Essential information required for user successCommands in a JEA session are often run with elevated privileges.

The following list contains examples of commands that can be used maliciously if allowed in an unconstrained state. This isn't an exhaustive list and should only be used as a cautionary starting point.

- Risk: Granting the connecting user admin privileges to bypass JEA

Example:

```
PowerShell  
  
Add-LocalGroupMember -Member 'CONTOSO\jdoe' -Group 'Administrators'
```

Related commands:

- Add-ADGroupMember
 - Add-LocalGroupMember
 - net.exe
 - dsadd.exe
- Risk: Running arbitrary code, such as malware, exploits, or custom scripts to bypass protections

Example:

```
PowerShell  
  
Start-Process -FilePath '\\san\share\malware.exe'
```

Related commands:

- Start-Process
- New-Service
- Invoke-Item
- Invoke-WmiMethod
- Invoke-CimMethod
- Invoke-Expression
- Invoke-Command
- New-ScheduledTask
- Register-ScheduledJob

Create a role capability file

You can create a new PowerShell role capability file with the [New-PSRoleCapabilityFile](#) cmdlet.

```
PowerShell
```

```
New-PSRoleCapabilityFile -Path .\MyFirstJEARole.psrc
```

You should edit the created role capability file to allow only the commands required for the role. The PowerShell help documentation contains several examples of how you can configure the file.

Allowing PowerShell cmdlets and functions

To authorize users to run PowerShell cmdlets or functions, add the cmdlet or function name to the **VisibleCmdlets** or **VisibleFunctions** fields. If you aren't sure whether a command is a cmdlet or function, you can run `Get-Command <name>` and check the **CommandType** property in the output.

```
PowerShell
```

```
VisibleCmdlets = @('Restart-Computer', 'Get-NetIPAddress')
```

Sometimes the scope of a specific cmdlet or function is too broad for your users' needs. A DNS admin, for example, may only need access to restart the DNS service. In multi-tenant environments, tenants have access to self-service management tools. Tenants should be limited to managing their own resources. For these cases, you can restrict which parameters are exposed from the cmdlet or function.

```
PowerShell
```

```
VisibleCmdlets = @{
    Name      = 'Restart-Computer'
    Parameters = @{ Name = 'Name' }
}
```

In more advanced scenarios, you may also need to restrict the values a user may use with these parameters. Role capabilities let you define a set of values or a regular expression pattern that determine what input is allowed.

```
PowerShell
```

```
VisibleCmdlets = @(
    @{
        Name      = 'Restart-Service'
        Parameters = @{ Name = 'Name'; ValidateSet = @('Dns', 'Spooler') }
    }
    @{
        Name      = 'Start-Website'
        Parameters = @{ Name = 'Name'; ValidatePattern = 'HR_*' }
    }
)
```

ⓘ Note

The [common PowerShell parameters](#) are always allowed, even if you restrict the available parameters. You shouldn't explicitly list them in the `Parameters` field.

The list below describes the various ways you can customize a visible cmdlet or function. You can mix and match any of the below in the `VisibleCmdlets` field.

- **Use case:** Allow the user to run `My-Func` without any restrictions on the parameters.

```
PowerShell
@{ Name = 'My-Func' }
```

- **Use case:** Allow the user to run `My-Func` from the module `MyModule` without any restrictions on the parameters.

```
PowerShell
@{ Name = 'MyModule\My-Func' }
```

- **Use case:** Allow the user to run any cmdlet or function with the verb `My`.

```
PowerShell
@{ Name = 'My-*' }
```

- **Use case:** Allow the user to run any cmdlet or function with the noun `Func`.

```
PowerShell
```

```
@{ Name = '*-Func' }
```

- **Use case:** Allow the user to run `My-Func` with the `Param1` and `Param2` parameters. Any value can be supplied to the parameters.

PowerShell

```
@{ Name = 'My-Func'; Parameters = @{ Name = 'Param1'}, @{ Name = 'Param2' } }
```

- **Use case:** Allow the user to run `My-Func` with the `Param1` parameter. Only `Value1` and `Value2` can be supplied to the parameter.

PowerShell

```
@{
    Name      = 'My-Func'
    Parameters = @{ Name = 'Param1'; ValidateSet = @('Value1',
    'Value2') }
}
```

- **Use case:** Allow the user to run `My-Func` with the `Param1` parameter. Any value starting with `contoso` can be supplied to the parameter.

PowerShell

```
@{
    Name      = 'My-Func'
    Parameters = @{ Name = 'Param1'; ValidatePattern = 'contoso.*' }
}
```

⚠ Warning

For best security practices, it isn't recommended to use wildcards when defining visible cmdlets or functions. Instead, you should explicitly list each trusted command to ensure no other commands that share the same naming scheme are unintentionally authorized.

You can't apply both a `ValidatePattern` and `ValidateSet` to the same cmdlet or function.

If you do, the `ValidatePattern` overrides the `ValidateSet`.

For more information about `ValidatePattern`, check out [this Hey, Scripting Guy! post](#) and the [PowerShell Regular Expressions](#) reference content.

Allowing external commands and PowerShell scripts

To allow users to run executables and PowerShell scripts (`.ps1`) in a JEA session, you have to add the full path to each program in the `VisibleExternalCommands` field.

PowerShell

```
VisibleExternalCommands = @(
    'C:\Windows\System32\whoami.exe'
    'C:\Program Files\Contoso\Scripts\UpdateITSoftware.ps1'
)
```

Where possible, use PowerShell cmdlet or function equivalents for any external executables you authorize since you have control over the parameters allowed with PowerShell cmdlets and functions.

Many executables allow you to read the current state and then change it by providing different parameters.

For example, consider the role of a file server admin that manages network shares hosted on a system. One way of managing shares is to use `net share`. However, allowing `net.exe` is dangerous because the user could use the command to gain admin privileges with the command `net group Administrators unprivilegedjeauser /add`. A more secure option is to allow the `Get-SmbShare` cmdlet, which achieves the same result but has a much more limited scope.

When making external commands available to users in a JEA session, always specify the complete path to the executable. This prevents the execution of similarly named and potentially malicious programs located elsewhere on the system.

Allowing access to PowerShell providers

By default, no PowerShell providers are available in JEA sessions. This reduces the risk of sensitive information and configuration settings being disclosed to the connecting user.

When necessary, you can allow access to the PowerShell providers using the `VisibleProviders` command. For a full list of providers, run `Get-PSProvider`.

PowerShell

```
VisibleProviders = 'Registry'
```

For simple tasks that require access to the file system, registry, certificate store, or other sensitive providers, consider writing a custom function that works with the provider on the user's behalf. The functions, cmdlets, and external programs available in a JEA session aren't subject to the same constraints as JEA. They can access any provider by default. Also consider using the [user drive](#) when users need to copy files to or from a JEA endpoint.

Creating custom functions

You can author custom functions in a role capability file to simplify complex tasks for your end users. Custom functions are also useful when you require advanced validation logic for cmdlet parameter values. You can write simple functions in the **FunctionDefinitions** field:

PowerShell

```
VisibleFunctions = 'Get-TopProcess'

FunctionDefinitions = @{
    Name      = 'Get-TopProcess'
    ScriptBlock = {
        param($Count = 10)

        Get-Process |
            Sort-Object -Property CPU -Descending |
            Microsoft.PowerShell.Utility\Select-Object -First $Count
    }
}
```

Important

Don't forget to add the name of your custom functions to the **VisibleFunctions** field so they can be run by the JEA users.

The body (script block) of custom functions runs in the default language mode for the system and isn't subject to JEA's language constraints. This means that functions can access the file system and registry, and run commands that weren't made visible in the role capability file. Take care to avoid running arbitrary code when using parameters. Avoid piping user input directly into cmdlets like `Invoke-Expression`.

In the above example, notice that the fully qualified module name (FQMN) `Microsoft.PowerShell.Utility\Select-Object` was used instead of the shorthand `Select-Object`. Functions defined in role capability files are still subject to the scope of JEA sessions, which includes the proxy functions JEA creates to constrain existing commands.

By default, `Select-Object` is a constrained cmdlet in all JEA sessions that doesn't allow the selection of arbitrary properties on objects. To use the unconstrained `Select-Object` in functions, you must explicitly request the full implementation using the FQMN. Any constrained cmdlet in a JEA session has the same constraints when invoked from a function. For more information, see [about_Command_Precedence](#).

If you're writing several custom functions, it's more convenient to put them in a PowerShell script module. You make those functions visible in the JEA session using the `VisibleFunctions` field like you would with built-in and third-party modules.

For tab completion to work properly in JEA sessions you must include the built-in function `TabExpansion2` in the `VisibleFunctions` list.

Make the role capabilities available to a configuration

Prior to PowerShell 6, for PowerShell to find a role capability file it must be stored in a `RoleCapabilities` folder in a PowerShell module. The module can be stored in any folder included in the `$Env:PSModulePath` environment variable, however you shouldn't place it in `$Env:SystemRoot\System32` or a folder where untrusted users could modify the files.

The following example creates a PowerShell script module called `ContosoJEA` in the `$Env:ProgramFiles` path to host the role capabilities file.

```
PowerShell

# Create a folder for the module
$modulePath = Join-Path $Env:ProgramFiles
"WindowsPowerShell\Modules\ContosoJEA"
New-Item -ItemType Directory -Path $modulePath

# Create an empty script module and module manifest.
# At least one file in the module folder must have the same name as the
# folder itself.
$rootModulePath = Join-Path $modulePath "ContosoJEAFunctions.psm1"
$moduleManifestPath = Join-Path $modulePath "ContosoJEA.psdi"
New-Item -ItemType File -Path $RootModulePath
```

```
New-ModuleManifest -Path $moduleManifestPath -RootModule  
"ContosoJEAFunctions.psm1"  
  
# Create the RoleCapabilities folder and copy in the PSRC file  
$rcFolder = Join-Path $modulePath "RoleCapabilities"  
New-Item -ItemType Directory $rcFolder  
Copy-Item -Path .\MyFirstJEARole.ps1 -Destination $rcFolder
```

For more information about PowerShell modules, see [Understanding a PowerShell Module](#).

Starting in PowerShell 6, the **RoleDefinitions** property was added to the session configuration file. This property lets you specify the location of a role configuration file for your role definition. See the examples in [New-PSSessionConfigurationFile](#).

Updating role capabilities

You can edit a role capability file to update the settings at any time. Any new JEA sessions started after the role capability has been updated will reflect the revised capabilities.

This is why controlling access to the role capabilities folder is so important. Only highly trusted administrators should be allowed to change role capability files. If an untrusted user can change role capability files, they can easily give themselves access to cmdlets that allow them to elevate their privileges.

For administrators looking to lock down access to the role capabilities, ensure Local System has read-only access to the role capability files and containing modules.

How role capabilities are merged

Users are granted access to all matching role capabilities in the [session configuration file](#) when they enter a JEA session. JEA tries to give the user the most permissive set of commands allowed by any of the roles.

VisibleCmdlets and VisibleFunctions

The most complex merge logic affects cmdlets and functions, which can have their parameters and parameter values limited in JEA.

The rules are as follows:

1. If a cmdlet is only made visible in one role, it's visible to the user with any applicable parameter constraints.
2. If a cmdlet is made visible in more than one role, and each role has the same constraints on the cmdlet, the cmdlet is visible to the user with those constraints.
3. If a cmdlet is made visible in more than one role, and each role allows a different set of parameters, the cmdlet and all the parameters defined across every role are visible to the user. If one role doesn't have constraints on the parameters, all parameters are allowed.
4. If one role defines a validate set or validate pattern for a cmdlet parameter, and the other role allows the parameter but doesn't constrain the parameter values, the validate set or pattern is ignored.
5. If a validate set is defined for the same cmdlet parameter in more than one role, all values from all validate sets are allowed.
6. If a validate pattern is defined for the same cmdlet parameter in more than one role, any values that match any of the patterns are allowed.
7. If a validate set is defined in one or more roles, and a validate pattern is defined in another role for the same cmdlet parameter, the validate set is ignored and rule (6) applies to the remaining validate patterns.

Below is an example of how roles are merged according to these rules:

PowerShell

```
# Role A Visible Cmdlets
$roleA = @{
    VisibleCmdlets = @(
        'Get-Service'
        @{
            Name      = 'Restart-Service'
            Parameters = @{
                Name = 'DisplayName'; ValidateSet = 'DNS Client'
            }
        }
    )
}

# Role B Visible Cmdlets
$roleB = @{
    VisibleCmdlets = @(
        @{
            Name      = 'Get-Service';
            Parameters = @{
                Name = 'DisplayName'; ValidatePattern = 'DNS.*'
            }
        }
        @{
            Name      = 'Restart-Service'
            Parameters = @{
                Name = 'DisplayName'; ValidateSet = 'DNS Server'
            }
        }
    )
}
```

```

        )
    }

# Resulting permissions for a user who belongs to both role A and B
# - The constraint in role B for the DisplayName parameter on Get-Service
#   is ignored because of rule #4
# - The ValidateSets for Restart-Service are merged because both roles use
#   ValidateSet on the same parameter per rule #5
$mergedAandB = @{
    VisibleCmdlets = @(
        'Get-Service'
        @{
            Name = 'Restart-Service';
            Parameters = @{
                Name = 'DisplayName'
                ValidateSet = 'DNS Client', 'DNS Server'
            }
        }
    )
}

```

VisibleExternalCommands, VisibleAliases, VisibleProviders, ScriptsToProcess

All other fields in the role capability file are added to a cumulative set of allowable external commands, aliases, providers, and startup scripts. Any command, alias, provider, or script available in one role capability is available to the JEA user.

Be careful to ensure that the combined set of providers from one role capability and cmdlets/functions/commands from another don't allow users unintentional access to system resources. For example, if one role allows the `Remove-Item` cmdlet and another allows the `FileSystem` provider, you are at risk of a JEA user deleting arbitrary files on your computer. Additional information about identifying users' effective permissions can be found in the [auditing JEA](#) article.

Next steps

[Create a session configuration file](#)

JEA Session Configurations

Article • 04/01/2024

A JEA endpoint is registered on a system by creating and registering a PowerShell session configuration file. Session configurations define who can use the JEA endpoint and which roles they have access to. They also define global settings that apply to all users of the JEA session.

Create a session configuration file

To register a JEA endpoint, you must specify how that endpoint is configured. There are many options to consider. The most important options are:

- Who has access to the JEA endpoint
- Which roles they may be assigned
- Which identity JEA uses under the covers
- The name of the JEA endpoint

These options are defined in a PowerShell data file with a `.pssc` extension known as a PowerShell session configuration file. The session configuration file can be edited using any text editor.

Run the following command to create a blank template configuration file.

PowerShell

```
New-PSSessionConfigurationFile -SessionType RestrictedRemoteServer -Path  
.\\MyJEAEndpoint.pssc
```

Tip

Only the most common configuration options are included in the template file by default. Use the `-Full` switch to include all applicable settings in the generated PSSC.

The `-SessionType RestrictedRemoteServer` field indicates that the session configuration is used by JEA for secure management. Sessions of this type operate in **NoLanguage** mode and only have access to the following default commands (and aliases):

- `Clear-Host` (`cls`, `clear`)

- `Exit-PSSession` (`exsn`, `exit`)
- `Get-Command` (`gcm`)
- `Get-FormatData`
- `Get-Help`
- `Measure-Object` (`measure`)
- `Out-Default`
- `Select-Object` (`select`)

No PowerShell providers are available, nor are any external programs (executables or scripts).

For more information about language modes, see [about_Language_Modes](#).

Choose the JEA identity

Behind the scenes, JEA needs an identity (account) to use when running a connected user's commands. You define which identity JEA uses in the session configuration file.

Local Virtual Account

Local virtual accounts are useful when all roles defined for the JEA endpoint are used to manage the local machine and a local administrator account is sufficient to run the commands successfully. Virtual accounts are temporary accounts that are unique to a specific user and only last for the duration of their PowerShell session. On a member server or workstation, virtual accounts belong to the local computer's **Administrators** group. On an Active Directory Domain Controller, virtual accounts belong to the domain's **Domain Admins** group.

```
PowerShell

# Setting the session to use a virtual account
RunAsVirtualAccount = $true
```

If the roles defined by the session configuration don't require full administrative privilege, you can specify the security groups to which the virtual account will belong. On a member server or workstation, the specified security groups must be local groups, not groups from a domain.

When one or more security groups are specified, the virtual account isn't assigned to the local or domain administrators group.

```
PowerShell
```

```
# Setting the session to use a virtual account that only belongs to the
NetworkOperator and NetworkAuditor local groups
RunAsVirtualAccount = $true
RunAsVirtualAccountGroups = 'NetworkOperator', 'NetworkAuditor'
```

ⓘ Note

Virtual accounts are temporarily granted the Logon as a service right in the local server security policy. If one of the VirtualAccountGroups specified has already been granted this right in the policy, the individual virtual account will no longer be added and removed from the policy. This can be useful in scenarios such as domain controllers where revisions to the domain controller security policy are closely audited. This is only available in Windows Server 2016 with the November 2018 or later rollup and Windows Server 2019 with the January 2019 or later rollup.

Group-managed service account

A group-managed service account (GMSA) is the appropriate identity to use when JEA users need to access network resources such as file shares and web services. GMSAs give you a domain identity that's used to authenticate with resources on any machine within the domain. The rights that a GMSA provides are determined by the resources you're accessing. You don't have admin rights on any machines or services unless the machine or service administrator has explicitly granted those rights to the GMSA.

PowerShell

```
# Configure JEA sessions to use the GMSA in the local computer's domain
# with the sAMAccountName of 'MyJEAGMSA'
GroupManagedServiceAccount = 'Domain\MyJEAGMSA'
```

GMSAs should only be used when necessary:

- It's difficult to trace back actions to a user when using a GMSA. Every user shares the same run-as identity. You must review PowerShell session transcripts and logs to correlate individual users with their actions.
- The GMSA may have access to many network resources that the connecting user doesn't need access to. Always try to limit effective permissions in a JEA session to follow the principle of least privilege.

ⓘ Note

Group managed service accounts are only available on domain-joined machines using PowerShell 5.1 or newer.

For more information about securing a JEA session, see the [security considerations](#) article.

Session transcripts

It's recommended that you configure a JEA endpoint to automatically record transcripts of users' sessions. PowerShell session transcripts contain information about the connecting user, the run as identity assigned to them, and the commands run by the user. They can be useful to an auditing team who needs to understand who made a specific change to a system.

To configure automatic transcription in the session configuration file, provide a path to a folder where the transcripts should be stored.

PowerShell

```
TranscriptDirectory = 'C:\ProgramData\JEAConfiguration\Transcripts'
```

Transcripts are written to the folder by the **Local System** account, which requires read and write access to the directory. Standard users should have no access to the folder. Limit the number of security administrators that have access to audit the transcripts.

User drive

If your connecting users need to copy files to or from the JEA endpoint, you can enable the user drive in the session configuration file. The user drive is a **PSDrive** that's mapped to a unique folder for each connecting user. This folder allows users to copy files to or from the system without giving them access to the full file system or exposing the **FileSystem** provider. The user drive contents are persistent across sessions to accommodate situations where network connectivity may be interrupted.

PowerShell

```
MountUserDrive = $true
```

By default, the user drive allows you to store a maximum of 50MB of data per user. You can limit the amount of data a user can consume with the *UserDriveMaximumSize* field.

PowerShell

```
# Enables the user drive with a per-user limit of 500MB (524288000 bytes)
MountUserDrive = $true
UserDriveMaximumSize = 524288000
```

If you don't want data in the user drive to be persistent, you can configure a scheduled task on the system to automatically clean up the folder every night.

 **Note**

The user drive is only available in PowerShell 5.1 or newer.

For more information about PSDrives, see [Managing PowerShell drives](#).

Role definitions

Role definitions in a session configuration file define the mapping of **users** to **roles**. Every user or group included in this field is granted permission to the JEA endpoint when it's registered. Each user or group can be included as a key in the hashtable only once, but can be assigned multiple roles. The name of the role capability should be the name of the role capability file, without the `.psrc` extension.

PowerShell

```
RoleDefinitions = @{
    'CONTOSO\JEA_DNS_ADMIN'      = @{ RoleCapabilities = 'DnsAdmin',
    'DnsOperator', 'DnsAuditor' }
    'CONTOSO\JEA_DNS_OPERATORS' = @{ RoleCapabilities = 'DnsOperator',
    'DnsAuditor' }
    'CONTOSO\JEA_DNS_AUDITORS' = @{ RoleCapabilities = 'DnsAuditor' }
}
```

If a user belongs to more than one group in the role definition, they get access to the roles of each. When two roles grant access to the same cmdlets, the most permissive parameter set is granted to the user.

When specifying local users or groups in the role definitions field, be sure to use the computer name, not **localhost** or wildcards. You can check the computer name by inspecting the `$Env:COMPUTERNAME` variable.

PowerShell

```
RoleDefinitions = @{
    'MyComputerName\MyLocalGroup' = @{
        RoleCapabilities = 'DnsAuditor'
    }
}
```

Role capability search order

As shown in the example above, role capabilities are referenced by the base name of the role capability file. The base name of a file is the filename without the extension. If multiple role capabilities are available on the system with the same name, PowerShell uses its implicit search order to select the effective role capability file. JEA does **not** give access to all role capability files with the same name.

JEA uses the `$Env:PSModulePath` environment variable to determine which paths to scan for role capability files. Within each of those paths, JEA looks for valid PowerShell modules that contain a "RoleCapabilities" subfolder. As with importing modules, JEA prefers role capabilities that are shipped with Windows to custom role capabilities with the same name.

For all other naming conflicts, precedence is determined by the order in which Windows enumerates the files in the directory. The order isn't guaranteed to be alphabetical. The first role capability file found that matches the specified name is used for the connecting user. Since the role capability search order isn't deterministic, it's **strongly recommended** that role capabilities have unique filenames.

Conditional access rules

All users and groups included in the **RoleDefinitions** field are automatically granted access to JEA endpoints. Conditional access rules allow you to refine this access and require users to belong to additional security groups that don't impact the roles to which they're assigned. This is useful when you want to integrate a just-in-time privileged access management solution, smartcard authentication, or other multifactor authentication solution with JEA.

Conditional access rules are defined in the **RequiredGroups** field in a session configuration file. There, you can provide a hashtable (optionally nested) that uses 'And' and 'Or' keys to construct your rules. Here are some examples of how to use this field:

PowerShell

```
# Example 1: Connecting users must belong to a security group called
#"elevated-jea"
RequiredGroups = @{ And = 'elevated-jea' }
```

```
# Example 2: Connecting users must have signed on with 2 factor  
authentication or a smart card  
# The 2 factor authentication group name is "2FA-logon" and the smart card  
group  
# name is "smartcard-logon"  
RequiredGroups = @{ Or = '2FA-logon', 'smartcard-logon' }  
  
# Example 3: Connecting users must elevate into "elevated-jea" with their  
JIT system and  
# have logged on with 2FA or a smart card  
RequiredGroups = @{ And = 'elevated-jea', @{ Or = '2FA-logon', 'smartcard-  
logon' } }
```

ⓘ Note

Conditional access rules are only available in PowerShell 5.1 or newer.

Other properties

Session configuration files can also do everything a role capability file can do, just without the ability to give connecting users access to different commands. If you want to allow all users access to specific cmdlets, functions, or providers, you can do so right in the session configuration file. For a full list of supported properties in the session configuration file, run `Get-Help New-PSSessionConfigurationFile -Full`.

Testing a session configuration file

You can test a session configuration using the [Test-PSSessionConfigurationFile](#) cmdlet. It's recommended that you test your session configuration file if you've manually edited the `.pssc` file. Testing ensures the syntax is correct. If a session configuration file fails this test, it can't be registered on the system.

Sample session configuration file

The following example shows how to create and validate a session configuration for JEA. The role definitions are created and stored in the `$roles` variable for convenience and readability. it isn't a requirement to do so.

PowerShell

```
$roles = @{  
    'CONTOSO\JEA_DNS_ADMIN' = @{ RoleCapabilities = 'DnsAdmin',
```

```
'DnsOperator', 'DnsAuditor' }
    'CONTOSO\JEA_DNS_OPERATORS' = @{
        RoleCapabilities = 'DnsOperator',
        'DnsAuditor' }
        'CONTOSO\JEA_DNS_AUDITORS' = @{
            RoleCapabilities = 'DnsAuditor' }
    }

$parameters = @{
    SessionType = 'RestrictedRemoteServer'
    Path = '.\JEAConfig.pssc'
    RunAsVirtualAccount = $true
    TranscriptDirectory = 'C:\ProgramData\JEAConfiguration\Transcripts'
    RoleDefinitions = $roles
    RequiredGroups = @{
        Or = '2FA-logon', 'smartcard-logon' }
}
New-PSSessionConfigurationFile @parameters
Test-PSSessionConfigurationFile -Path .\JEAConfig.pssc # should yield True
```

Updating session configuration files

To change the properties of a JEA session configuration, including the mapping of users to roles, you must [unregister](#). Then, [re-register](#) the JEA session configuration using an updated session configuration file.

Next steps

- [Register a JEA configuration](#)
- [Author JEA roles](#)

Registering JEA Configurations

Article • 04/01/2024

Once you have your [role capabilities](#) and [session configuration file](#) created, the last step is to register the JEA endpoint. Registering the JEA endpoint with the system makes the endpoint available for use by users and automation engines.

Single machine configuration

For small environments, you can deploy JEA by registering the session configuration file using the [Register-PSSessionConfiguration](#) cmdlet.

Before you begin, ensure that the following prerequisites have been met:

- One or more roles has been created and placed in the **RoleCapabilities** folder of a PowerShell module.
- A session configuration file has been created and tested.
- The user registering the JEA configuration has administrator rights on the system.
- You've selected a name for your JEA endpoint.

The name of the JEA endpoint is required when users connect to the system using JEA. The [Get-PSSessionConfiguration](#) cmdlet lists the names of the endpoints on a system. Endpoints that start with `microsoft` are typically shipped with Windows. The `microsoft.powershell` endpoint is the default endpoint used when connecting to a remote PowerShell endpoint.

PowerShell

```
Get-PSSessionConfiguration | Select-Object Name
```

Output

Name

<code>microsoft.powershell</code>
<code>microsoft.powershell.workflow</code>
<code>microsoft.powershell132</code>

Run the following command to register the endpoint.

PowerShell

```
Register-PSSessionConfiguration -Path .\MyJEAConfig.pssc -Name  
'JEA Maintenance' -Force
```

⚠️ Warning

The previous command restarts the WinRM service on the system. This terminates all PowerShell remoting sessions and any ongoing DSC configurations. We recommended you take production machines offline before running the command to avoid disrupting business operations.

After registration, you're ready to [use JEA](#). You may delete the session configuration file at any time. The configuration file isn't used after registration of the endpoint.

Multi-machine configuration with DSC

When deploying JEA on multiple machines, the simplest deployment model uses the [JEA Desired State Configuration \(DSC\)](#) resource to quickly and consistently deploy JEA on each machine.

To deploy JEA with DSC, ensure the following prerequisites are met:

- One or more role capabilities have been authored and added to a PowerShell module.
- The PowerShell module containing the roles is stored on a (read-only) file share accessible by each machine.
- Settings for the session configuration have been determined. You don't need to create a session configuration file when using the JEA DSC resource.
- You have credentials that allow administrative actions on each machine or access to the DSC pull server used to manage the machines.
- You've downloaded the [JEA DSC resource ↗](#).

Create a DSC configuration for your JEA endpoint on a target machine or pull server. In this configuration, the **JustEnoughAdministration** DSC resource defines the session configuration file and the **File** resource copies the role capabilities from the file share.

The following properties are configurable using the DSC resource:

- Role Definitions
- Virtual account groups
- Group-managed service account name
- Transcript directory

- User drive
- Conditional access rules
- Startup scripts for the JEA session

The syntax for each of these properties in a DSC configuration is consistent with the PowerShell session configuration file.

Below is a sample DSC configuration for a general server maintenance module. It assumes that a valid PowerShell module containing role capabilities is located on the `\myfileshare\JEA` file share.

```
PowerShell

Configuration JEAMaintenance
{
    Import-DscResource -Module JustEnoughAdministration,
    PSDesiredStateConfiguration

    File MaintenanceModule
    {
        SourcePath = "\myfileshare\JEA\ContosoMaintenance"
        DestinationPath = "C:\Program
Files\WindowsPowerShell\Modules\ContosoMaintenance"
        Checksum = "SHA-256"
        Ensure = "Present"
        Type = "Directory"
        Recurse = $true
    }

    JeaEndpoint JEAMaintenanceEndpoint
    {
        EndpointName = "JEAMaintenance"
        RoleDefinitions = "@{ 'CONTOSO\JEAMaintenanceAuditors' = @{
RoleCapabilities = 'GeneralServerMaintenance-Audit' };
'CONTOSO\JEAMaintenanceAdmins' = @{ RoleCapabilities =
'GeneralServerMaintenance-Audit', 'GeneralServerMaintenance-Admin' } }"
        TranscriptDirectory = 'C:\ProgramData\JEAConfiguration\Transcripts'
        DependsOn = '[File]MaintenanceModule'
    }
}
```

Next, the configuration is applied on a system by directly invoking the [Local Configuration Manager](#) or updating the [pull server configuration](#).

The DSC resource also allows you to replace the default **Microsoft.PowerShell** endpoint. When replaced, the resource automatically registers a backup endpoint named **Microsoft.PowerShell.Restricted**. The backup endpoint has the default WinRM ACL that allows Remote Management Users and local Administrators group members to access it.

Unregistering JEA configurations

The [Unregister-PSSessionConfiguration](#) cmdlet removes a JEA endpoint. Unregistering a JEA endpoint prevents new users from creating new JEA sessions on the system. It also allows you to update a JEA configuration by re-registering an updated session configuration file using the same endpoint name.

PowerShell

```
# Unregister the JEA endpoint called "ContosoMaintenance"
Unregister-PSSessionConfiguration -Name 'ContosoMaintenance' -Force
```

⚠ Warning

Unregistering a JEA endpoint causes the WinRM service to restart. This interrupts most remote management operations in progress, including other PowerShell sessions, WMI invocations, and some management tools. Only unregister PowerShell endpoints during planned maintenance windows.

Next steps

[Test the JEA endpoint](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Using JEA

Article • 04/01/2024

This article describes the various ways you can connect to and use a JEA endpoint.

Using JEA interactively

If you're testing your JEA configuration or have simple tasks for users, you can use JEA the same way you would a regular PowerShell remoting session. For complex remoting tasks, it's recommended to use [implicit remoting](#). Implicit remoting allows users to operate with the data objects locally.

To use JEA interactively, you need:

- The name of the computer you're connecting to (can be the local machine)
- The name of the JEA endpoint registered on that computer
- Credentials that have access to the JEA endpoint on that computer

Given that information, you can start a JEA session using the [New-PSSession](#) or [Enter-PSSession](#) cmdlets.

```
PowerShell

$sessionParams = @{
    ComputerName      = 'localhost'
    ConfigurationName = 'JEAMaintenance'
    Credential        = Get-Credential
}
Enter-PSSession @sessionParams
```

If the current user account has access to the JEA endpoint, you can omit the **Credential** parameter.

When the PowerShell prompt changes to `[localhost]: PS>` you know that you're now interacting with the remote JEA session. You can run `Get-Command` to check which commands are available. Consult with your administrator to learn if there are any restrictions on the available parameters or allowed parameter values.

Remember, JEA sessions operate in `NoLanguage` mode. Some of the ways you typically use PowerShell may not be available. For instance, you can't use variables to store data or inspect the properties on objects returned from cmdlets. The following example shows two approaches to get the same commands to work in `NoLanguage` mode.

PowerShell

```
# Using variables is prohibited in NoLanguage mode. The following will not work:  
# $vm = Get-VM -Name 'SQL01'  
# Start-VM -VM $vm  
  
# You can use pipes to pass data through to commands that accept input from the pipeline  
Get-VM -Name 'SQL01' | Start-VM  
  
# You can also wrap subcommands in parentheses and enter them inline as arguments  
Start-VM -VM (Get-VM -Name 'SQL01')  
  
# You can also use parameter sets that don't require extra data to be passed in  
Start-VM -VMName 'SQL01'
```

For more complex command invocations that make this approach difficult, consider using [implicit remoting](#) or [creating custom functions](#) that wrap the functionality you require. For more information on `NoLanguageMode`, see [about_Language_Modes](#).

Using JEA with implicit remoting

PowerShell has an implicit remoting model that lets you import proxy cmdlets from a remote machine and interact with them as if they were local commands. Implicit remoting is explained in this *Hey, Scripting Guy!* [blog post](#). Implicit remoting is useful when working with JEA because it allows you to work with JEA cmdlets in a full language mode. You can use tab completion, variables, manipulate objects, and even use local scripts to automate tasks against a JEA endpoint. Anytime you invoke a proxy command, the data is sent to the JEA endpoint on the remote machine and executed there.

Implicit remoting works by importing cmdlets from an existing PowerShell session. You can optionally choose to prefix the nouns of each proxy cmdlet with a string of your choosing. The prefix allows you to distinguish the commands that are for the remote system. A temporary script module containing all the proxy commands is created and imported for the duration of your local PowerShell session.

PowerShell

```
# Create a new PSSession to your JEA endpoint  
$jeaSession = New-PSSession -ComputerName 'SERVER01' -ConfigurationName 'JEA  
Maintenance'  
  
# Import the entire PSSession and prefix each imported cmdlet with "JEA"
```

```
Import-PSSession -Session $jeaSession -Prefix 'JEA'

# Invoke "Get-Command" on the remote JEA endpoint using the proxy cmdlet
Get-JEACmdlet
```

ⓘ Important

Some systems may not be able to import an entire JEA session due to constraints in the default JEA cmdlets. To get around this, only import the commands you need from the JEA session by explicitly providing their names to the `-CommandName` parameter. A future update will address the issue with importing entire JEA sessions on affected systems.

If you're unable to import a JEA session because of JEA constraints on the default parameters, follow the steps below to filter out the default commands from the imported set. You can continue use commands like `Select-Object`, but you'll just use the local version installed on your computer instead of the one imported from the remote JEA session.

PowerShell

```
# Create a new PSSession to your JEA endpoint
$jeaSession = New-PSSession -ComputerName 'SERVER01' -ConfigurationName
'JEMaintenance'

# Get a list of all the commands on the JEA endpoint
$commands = Invoke-Command -Session $jeaSession -ScriptBlock { Get-Command }

# Filter out the default cmdlets
$jeaDefaultCmdlets = @(
    'Clear-Host'
    'Exit-PSSession'
    'Get-Command'
    'Get-FormatData'
    'Get-Help'
    'Measure-Object'
    'Out-Default'
    'Select-Object'
)
$filteredCommands = $commands | Where-Object { $jeaDefaultCmdlets -
notcontains $_ }

# Import only commands explicitly added in role capabilities and prefix each
# imported cmdlet with "JEA"
Import-PSSession -Session $jeaSession -Prefix 'JEA' -CommandName
$filteredCommands
```

You can also persist the proxied cmdlets from implicit remoting using [Export-PSSession](#). For more information about implicit remoting, see the documentation for [Import-PSSession](#) and [Import-Module](#).

Using JEA programmatically

JEA can also be used in automation systems and in user applications, such as in-house helpdesk apps and websites. The approach is the same as that for building apps that talk to unconstrained PowerShell endpoints. Ensure the program is designed to work with limitation imposed by JEA.

For simple, one-off tasks, you can use [Invoke-Command](#) to run commands in a JEA session.

PowerShell

```
Invoke-Command -ComputerName 'SERVER01' -ConfigurationName 'JEAMaintenance'
-ScriptBlock {
    Get-Process
    Get-Service
}
```

To check which commands are available for use when you connect to a JEA session, run [Get-Command](#) and iterate through the results to check for the allowed parameters.

PowerShell

```
$commandParameters = @{
    ComputerName      = 'SERVER01'
    ConfigurationName = 'JEAMaintenance'
    ScriptBlock       = { Get-Command }
}
Invoke-Command @commandParameters |
    Where-Object { $_. CommandType -in @('Function', 'Cmdlet') } |
        Format-Table Name, Parameters
```

If you're building a C# app, you can create a PowerShell runspace that connects to a JEA session by specifying the configuration name in a [WSManConnectionInfo](#) object.

C#

```
// using System.Management.Automation;
var computerName = "SERVER01";
var configName   = "JEAMaintenance";
// See
https://learn.microsoft.com/dotnet/api/system.management.automation.pscreden
```

```

tial
var creds      = // create a PSCredential object here

WSManConnectionInfo connectionInfo = new WSManConnectionInfo(
    false,           // Use SSL
    computerName,   // Computer name
    5985,           // WSMAN Port
    "/wsman",       // WSMAN Path
    string.Format(  // Connection URI with config name
        CultureInfo.InvariantCulture,
        "http://schemas.microsoft.com/powershell/{0}",
        configName
    ),
    creds           // Credentials
);

// Now, use the connection info to create a runspace where you can run the
// commands
using (Runspace runspace = RunspaceFactory.CreateRunspace(connectionInfo))
{
    // Open the runspace
    runspace.Open();

    using (PowerShell ps = PowerShell.Create())
    {
        // Set the PowerShell object to use the JEA runspace
        ps.Runspace = runspace;

        // Now you can add and invoke commands
        ps.AddCommand("Get-Command");
        foreach (var result in ps.Invoke())
        {
            Console.WriteLine(result);
        }
    }

    // Close the runspace
    runspace.Close();
}

```

Using JEA with PowerShell Direct

Hyper-V in Windows 10 and Windows Server 2016 offers [PowerShell Direct](#), a feature that allows Hyper-V administrators to manage virtual machines with PowerShell regardless of the network configuration or remote management settings on the virtual machine.

You can use PowerShell Direct with JEA to give a Hyper-V administrator limited access to your VM. This can be useful if you lose network connectivity to your VM and need a

datacenter admin to fix the network settings.

No additional configuration is required to use JEA over PowerShell Direct. However, the guest operating system running inside the virtual machine must be Windows 10, Windows Server 2016, or higher. The Hyper-V admin can connect to the JEA endpoint by using the `-VMName` or `-VMId` parameters on PSRemoting cmdlets:

PowerShell

```
$sharedParams = @{
    ConfigurationName = 'NICMaintenance'
    Credential       = Get-Credential -UserName 'localhost\JEAformyHoster'
}
# Entering a JEA session using PowerShell Direct when the VM name is unique
Enter-PSSession -VMName 'SQL01' @sharedParams

# Entering a JEA session using PowerShell Direct using VM ids
$vm = Get-VM -VMName 'MyVM' | Select-Object -First 1
Enter-PSSession -VMId $vm.VMId @sharedParams
```

It's recommended you create a dedicated user account with the minimum rights needed to manage the system for use by a Hyper-V administrator. Remember, even an unprivileged user can sign into a Windows machine by default, including using unconstrained PowerShell. That allows them to browse the file system and learn more about your OS environment. To lock down a Hyper-V administrator and limit them to only access a VM using PowerShell Direct with JEA, you must deny local logon rights to the Hyper-V admin's JEA account.



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

JEA Security Considerations

Article • 04/01/2024

JEA helps you improve your security posture by reducing the number of permanent administrators on your machines. JEA uses a PowerShell session configuration to create a new entry point for users to manage the system. Users who need elevated, but not unlimited, access to the machine to do administrative tasks can be granted access to the JEA endpoint. Since JEA allows these users to run administrative commands without having full administrator access, you can then remove those users from highly privileged security groups.

Run-As account

Each JEA endpoint has a designated **run-as** account under which the connecting user's actions are executed. This account is configurable in the [session configuration file](#), and the account you choose has a significant bearing on the security of your endpoint.

Virtual accounts are the recommended way of configuring the **run-as** account. Virtual accounts are one-time, temporary local accounts that are created for the connecting user to use during the duration of their JEA session. As soon as their session is terminated, the virtual account is destroyed and can't be used anymore. The connecting user doesn't know the credentials for the virtual account. The virtual account can't be used to access the system via other means like Remote Desktop or an unconstrained PowerShell endpoint.

By default, virtual accounts are members of the local **Administrators** group on the machine. This membership gives them full rights to manage anything on the system, but no rights to manage resources on the network. When the user connects to other machines from the JEA session, the user context is that of the local computer account, not the virtual account.

Domain controllers are a special case since there isn't a local **Administrators** group. Instead, virtual accounts belong to **Domain Admins** and can manage the directory services on the domain controller. The domain identity is still restricted for use on the domain controller where the JEA session was instantiated. Any network access appears to come from the domain controller computer object instead.

In both cases, you may assign the virtual account to specific security groups, especially when the task can be done without local or domain administrator privileges. If you already have a security group defined for your administrators, grant the virtual account membership to that group. Group membership for virtual accounts is limited to local

security groups on workstation and member servers. On domain controllers, virtual accounts must be members of domain security groups. Once the virtual account has been added to one or more security groups, it no longer belongs to the default groups (local or domain administrators).

The following table summarizes the possible configuration options and resulting permissions for virtual accounts:

[+] Expand table

Computer type	Virtual account group configuration	Local user context	Network user context
Domain controller	Default	Domain user, member of <code><DOMAIN>\Domain Admins</code>	Computer account
Domain controller	Domain groups A and B	Domain user, member of <code><DOMAIN>\A, <DOMAIN>\B</code>	Computer account
Member server or workstation	Default	Local user, member of <code>BUILTIN\Administrators</code>	Computer account
Member server or workstation	Local groups C and D	Local user, member of <code><COMPUTER>\C and <COMPUTER>\D</code>	Computer account

When you look at Security audit and Application event logs, you see that each JEA user session has a unique virtual account. This unique account helps you track user actions in a JEA endpoint back to the original user who ran the command. Virtual account names follow the format `WinRM Virtual`

`Users\WinRM_VA_<ACCOUNTNUMBER>_<DOMAIN>_<sAMAccountName>` For example, if user **Alice** in domain **Contoso** restarts a service in a JEA endpoint, the username associated with any service control manager events would be `WinRM Virtual`

`Users\WinRM_VA_1_contoso_alice.`

Group-managed service accounts (gMSAs) are useful when a member server needs to have access to network resources in the JEA session. For example, when a JEA endpoint is used to control access to a REST API service hosted on a different machine. It's easy to write functions to invoke the REST APIs, but you need a network identity to authenticate with the API. Using a group-managed service account makes the second hop possible while maintaining control over which computers can use the account. The security group (local or domain) memberships of the gMSA defined the effective permissions for the gMSA account.

When a JEA endpoint is configured to use a gMSA, the actions of all JEA users appear to come from the same gMSA. The only way to trace actions back to a specific user is to identify the set of commands run in a PowerShell session transcript.

Pass-through credentials are used when you don't specify a **run-as** account. PowerShell uses the connecting user's credential to run commands on the remote server. To use pass-through credentials, you must grant the connecting user direct access to privileged management groups. This configuration is **NOT** recommended for JEA. If the connecting user already has administrator privileges, they can bypass JEA and manage the system using other access methods.

Standard run-as accounts allow you to specify any user account under which the entire PowerShell session runs. Session configurations using fixed **run-as** accounts (with the `-RunAsCredential` parameter) aren't JEA-aware. Role definitions no longer function as expected. Every user authorized to access the endpoint is assigned the same role.

You shouldn't use a **RunAsCredential** on a JEA endpoint because it's difficult to trace actions back to specific users and lacks support for mapping users to roles.

WinRM Endpoint ACL

As with regular PowerShell remoting endpoints, each JEA endpoint has a separate access control list (ACL) that controls who can authenticate with the JEA endpoint. If improperly configured, trusted users may not be able to access the JEA endpoint, and untrusted users may have access. The WinRM ACL doesn't affect the mapping of users to JEA roles. Mapping is controlled by the **RoleDefinitions** field in the session configuration file used to register the endpoint.

By default, when a JEA endpoint has multiple role capabilities, the WinRM ACL is configured to allow access to all mapped users. For example, a JEA session configured using the following commands grants full access to `CONTOSO\JEA_Lev1` and `CONTOSO\JEA_Lev2`.

PowerShell

```
$roles = @{ 'CONTOSO\JEA_Lev1' = 'Lev1Role'; 'CONTOSO\JEA_Lev2' = 'Lev2Role' }
New-PSSessionConfigurationFile -Path '.\jea.pssc' -SessionType
RestrictedRemoteServer -RoleDefinitions $roles -RunAsVirtualAccount
Register-PSSessionConfiguration -Path '.\jea.pssc' -Name 'MyJEAEndpoint'
```

You can audit user permissions with the [Get-PSSessionConfiguration](#) cmdlet.

PowerShell

```
Get-PSSessionConfiguration -Name 'MyJEAEndpoint' | Select-Object Permission
```

Output

```
Permission
```

```
-----  
CONTOSO\JEA_Lev1 AccessAllowed  
CONTOSO\JEA_Lev2 AccessAllowed
```

To change which users have access, run either `Set-PSSessionConfiguration -Name 'MyJEAEndpoint' -ShowSecurityDescriptorUI` for an interactive prompt or `Set-PSSessionConfiguration -Name 'MyJEAEndpoint' -SecurityDescriptorSddl <SDDL string>` to update the permissions. Users need at least *Invoke* rights to access the JEA endpoint.

It's possible to create a JEA endpoint that doesn't map a defined role to every user that has access. These users can start a JEA session, but only have access to the default cmdlets. You can audit user permissions in a JEA endpoint by running `Get-PSSessionCapability`. For more information, see [Auditing and Reporting on JEA](#).

Least privilege roles

When designing JEA roles, it's important to remember that the virtual and group-managed service accounts running behind the scenes can have unrestricted access to the local machine. JEA role capabilities help limit the commands and applications that can be run in that privileged context. Improperly designed roles can allow dangerous commands that may permit a user to break out of the JEA boundaries or obtain access to sensitive information.

For example, consider the following role capability entry:

PowerShell

```
@{  
    VisibleCmdlets = 'Microsoft.PowerShell.Management\*-Process'  
}
```

This role capability allows users to run any PowerShell cmdlet with the noun **Process** from the **Microsoft.PowerShell.Management** module. Users may need to access cmdlets like `Get-Process` to see what applications are running on the system and `Stop-Process` to kill applications that aren't responding. However, this entry also allows

`Start-Process`, which can be used to start up an arbitrary program with full administrator permissions. The program doesn't need to be installed locally on the system. A connected user could start a program from a file share that gives the user local administrator privileges, runs malware, and more.

A more secure version of this same role capability would look like:

```
PowerShell

#{@
    VisibleCmdlets = 'Microsoft.PowerShell.Management\Get-Process',
                      'Microsoft.PowerShell.Management\Stop-Process'
}
```

Avoid using wildcards in role capabilities. Be sure to regularly audit effective user permissions to see which commands are accessible to a user. For more information, see the *Check effective rights* section of the [Auditing and Reporting on JEA](#) article.

Best practice recommendations

The following are best practice recommendations to ensure the security of your JEA endpoints:

Limit the use and capabilities of PowerShell providers

Review how the allowed providers are used to ensure that you don't create vulnerabilities in your configured session.

Warning

Don't allow the **FileSystem** provider. If users can write to any part of the file system, it's possible to completely bypass security.

Don't allow the **Certificate** provider. With the provider enabled, a user could gain access to stored private keys.

Don't allow commands that can create new runspaces.

Warning

The `*-Job` cmdlets can create new runspaces without the restrictions.

Don't allow the `Trace-Command` cmdlet.

⚠ Warning

Using `Trace-Command` brings all traced commands into the session.

Don't create your own proxy implementations for the *restricted commands*.

PowerShell has a set of proxy commands for restricted command scenarios. These proxy commands ensure that input parameters can't compromise the security of the session.

The following commands have restricted proxies:

- `Exit-PSSession`
- `Get-Command`
- `Get-FormatData`
- `Get-Help`
- `Measure-Object`
- `Out-Default`
- `Select-Object`

If you create your own implementation of these commands, you may inadvertently allow users to run code prohibited by the JEA proxy commands.

JEA doesn't protect against admins

One of the core principles of JEA is that it allows nonadministrators to do some administrative tasks. JEA doesn't protect against users who already have administrator privileges. Users who belong **Domain Admins**, local **Administrators**, or other highly privileged groups can circumvent JEA's protections in other ways. For example, they could sign in with RDP, use remote MMC consoles, or connect to unconstrained PowerShell endpoints. Also, local administrator on a system can modify JEA configurations to add more users or change a role capability to extend the scope of what a user can do in their JEA session. It's important to evaluate your JEA users' extended permissions to see if there are other ways to gain privileged access to the system.

In addition to using JEA for regular day-to-day maintenance, it's common to have a just-in-time privileged access management system. These systems allow designated users to temporarily become a local administrator only after they complete a workflow that documents their use of those permissions.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Auditing and Reporting on JEA

Article • 04/01/2024

After you've deployed JEA, you need to regularly audit the JEA configuration. Auditing helps you assess that the correct people have access to the JEA endpoint and their assigned roles are still appropriate.

Find registered JEA sessions on a machine

To check which JEA sessions are registered on a machine, use the [Get-PSSessionConfiguration](#) cmdlet.

PowerShell

```
# Filter for sessions that are configured as 'RestrictedRemoteServer' to
# find JEA-like session configurations
Get-PSSessionConfiguration | Where-Object { $_.SessionType -eq
'RestrictedRemoteServer' }
```

Output

```
Name      : JEAMaintenance
PSVersion : 5.1
StartupScript :
RunAsUser  :
Permission : CONTOSO\JEA_DNS_ADMIN AccessAllowed,
CONTOSO\JEA_DNS_OPERATORS AccessAllowed,
CONTOSO\JEA_DNS_AUDITORS AccessAllowed
```

The effective rights for the endpoint are listed in the **Permission** property. These users have the right to connect to the JEA endpoint. However, the roles and commands they have access to is determined by the **RoleDefinitions** property in the [session configuration file](#) that was used to register the endpoint. Expand the **RoleDefinitions** property to evaluate the role mappings in a registered JEA endpoint.

PowerShell

```
# Get the desired session configuration
$jea = Get-PSSessionConfiguration -Name 'JEAMaintenance'

# Enumerate users/groups and which roles they have access to
$jea.RoleDefinitions.GetEnumerator() | Select-Object Name, @{
    Name = 'Role Capabilities'
```

```
Expression = { $_.Value.RoleCapabilities }  
}
```

Find available role capabilities on the machine

JEA gets role capabilities from the `.psrc` files stored in the `RoleCapabilities` folder inside a PowerShell module. The following function finds all role capabilities available on a computer.

PowerShell

```
function Find-LocalRoleCapability {  
    $results = @()  
  
    # Find modules with a "RoleCapabilities" subfolder and add any PSRC  
files to the result set  
    Get-Module -ListAvailable | ForEach-Object {  
        $psrcpath = Join-Path -Path $_.ModuleBase -ChildPath  
'RoleCapabilities'  
        if (Test-Path $psrcpath) {  
            $results += Get-ChildItem -Path $psrcpath -Filter *.psrc  
        }  
    }  
  
    # Format the results nicely to make it easier to read  
    $results | Select-Object @{ Name = 'Name'; Expression = {  
$__.Name.TrimEnd('.psrc') }}, @{  
    Name = 'Path'; Expression = { $_.FullName }  
} | Sort-Object Name  
}
```

ⓘ Note

The order of results from this function isn't necessarily the order in which the role capabilities will be selected if multiple role capabilities share the same name.

Check effective rights for a specific user

The `Get-PSSessionCapability` cmdlet enumerates all the commands available on a JEA endpoint based on a user's group memberships. The output of `Get-PSSessionCapability` is identical to that of the specified user running `Get-Command - CommandType All` in a JEA session.

PowerShell

```
Get-PSSessionCapability -ConfigurationName 'JEA Maintenance' -Username  
'CONTOSO\Alice'
```

If your users aren't permanent members of groups that would grant them additional JEA rights, this cmdlet may not reflect those extra permissions. This happens when using just-in-time privileged access management systems to allow users to temporarily belong to a security group. Carefully evaluate the mapping of users to roles and capabilities to ensure that users only get the level of access needed to do their jobs successfully.

PowerShell event logs

If you enabled module or script block logging on the system, you can see events in the Windows event logs for each command a user runs in a JEA session. To find these events, open **Microsoft-Windows-PowerShell/Operational** event log and look for events with event ID 4104.

Each event log entry includes information about the session in which the command was run. For JEA sessions, the event includes information about the **ConnectedUser** and the **RunAsUser**. The **ConnectedUser** is the actual user who created the JEA session. The **RunAsUser** is the account JEA used to execute the command.

Application event logs show changes being made by the **RunAsUser**. So having module and script logging enabled is required to trace a specific command invocation back to the **ConnectedUser**.

Application event logs

Commands run in a JEA session that interact with external applications or services may log events to their own event logs. Unlike PowerShell logs and transcripts, other logging mechanisms don't capture the connected user of the JEA session. Instead, those applications only log the virtual run-as user. To determine who ran the command, you need to consult a [session transcript](#) or correlate PowerShell event logs with the time and user shown in the application event log.

The WinRM log can also help you correlate run-as users to the connecting user in an application event log. Event ID 193 in the **Microsoft-Windows-Windows Remote Management/Operational** log records the security identifier (SID) and account name for both the connecting user and run as user for every new JEA session.

Session transcripts

If you configured JEA to create a transcript for each user session, a text copy of every user's actions are stored in the specified folder.

The following command (as an administrator) finds all transcript directories.

```
PowerShell
```

```
Get-PSSessionConfiguration |  
    Where-Object { $_.TranscriptDirectory -ne $null } |  
        Format-Table Name, TranscriptDirectory
```

Each transcript starts with information about the time the session started, which user connected to the session, and which JEA identity was assigned to them.

```
*****  
Windows PowerShell transcript start  
Start time: 20160710144736  
Username: CONTOSO\Alice  
RunAs User: WinRM Virtual Users\WinRM VA_1_CONTOSO_Alice  
Machine: SERVER01 (Microsoft Windows NT 10.0.14393.0)  
[...]
```

The body of the transcript contains information about each command the user invoked. The exact syntax of the command used is unavailable in JEA sessions because of the way commands are transformed for PowerShell remoting. However, you can still determine the effective command that was executed. Below is an example transcript snippet from a user running `Get-Service Dns` in a JEA session:

```
PS>CommandInvocation(Get-Service): "Get-Service"  
>> ParameterBinding(Get-Service): name="Name"; value="Dns"  
>> CommandInvocation(Out-Default): "Out-Default"  
>> ParameterBinding(Out-Default): name="InputObject"; value="Dns"  
  
Running   Dns           DNS Server
```

A **CommandInvocation** line is written for each command a user runs.

ParameterBindings record each parameter and value supplied with the command. In the previous example, you can see that the parameter **Name** was supplied the value **Dns** for the `Get-Service` cmdlet.

The output of each command also triggers a `CommandInvocation`, usually to `Out-Default`. The `InputObject` of `Out-Default` is the PowerShell object returned from the command. The details of that object are printed a few lines below, closely mimicking what the user would have seen.

See also

[PowerShell ❤ the Blue Team blog post on security ↗](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Running Remote Commands

Article • 03/24/2025

You can run commands on one or hundreds of computers with a single PowerShell command. Windows PowerShell supports remote computing using various technologies, including WMI, RPC, and WS-Management.

PowerShell supports WMI, WS-Management, and SSH remoting. In PowerShell 7 and higher, RPC is supported only on Windows.

For more information about remoting in PowerShell, see the following articles:

- [SSH Remoting in PowerShell](#)
- [WSMan Remoting in PowerShell](#)

Windows PowerShell remoting without configuration

Many Windows PowerShell cmdlets have the **ComputerName** parameter that enables you to collect data and change settings on one or more remote computers. These cmdlets use varying communication protocols and work on all Windows operating systems without any special configuration.

These cmdlets include:

- [Restart-Computer](#)
- [Test-Connection](#)
- [Clear-EventLog](#)
- [Get-EventLog](#)
- [Get-HotFix](#)
- [Get-Process](#)
- [Get-Service](#)
- [Set-Service](#)
- [Get-WinEvent](#)
- [Get-WmiObject](#)

Typically, cmdlets that support remoting without special configuration have the **ComputerName** parameter and don't have the **Session** parameter. To find these cmdlets in your session, type:

```
PowerShell
```

```
Get-Command | Where-Object {  
    $_.Parameters.Keys -contains "ComputerName" -and  
    $_.Parameters.Keys -notcontains "Session"  
}
```

Windows PowerShell remoting

Using the WS-Management protocol, Windows PowerShell remoting lets you run any Windows PowerShell command on one or more remote computers. You can establish persistent connections, start interactive sessions, and run scripts on remote computers.

To use Windows PowerShell remoting, the remote computer must be configured for remote management. For more information, including instructions, see [About Remote Requirements](#).

Once you have configured Windows PowerShell remoting, many remoting strategies are available to you. This article lists just a few of them. For more information, see [About Remote](#).

Start an interactive session

To start an interactive session with a single remote computer, use the [Enter-PSSession](#) cmdlet. For example, to start an interactive session with the Server01 remote computer, type:

```
PowerShell  
  
Enter-PSSession Server01
```

The command prompt changes to display the name of the remote computer. Any commands that you type at the prompt run on the remote computer and the results are displayed on the local computer.

To end the interactive session, type:

```
PowerShell  
  
Exit-PSSession
```

For more information about the [Enter-PSSession](#) and [Exit-PSSession](#) cmdlets, see:

- [Enter-PSSession](#)

- [Exit-PSSession](#)

Run a Remote Command

To run a command on one or more computers, use the [Invoke-Command](#) cmdlet. For example, to run a [Get-UICulture](#) command on the Server01 and Server02 remote computers, type:

```
PowerShell
```

```
Invoke-Command -ComputerName Server01, Server02 -ScriptBlock {Get-UICulture}
```

The output is returned to your computer.

```
Output
```

LCID	Name	DisplayName	PSComputerName
1033	en-US	English (United States)	server01.corp.fabrikam.com
1033	en-US	English (United States)	server02.corp.fabrikam.com

Run a Script

To run a script on one or many remote computers, use the **FilePath** parameter of the [Invoke-Command](#) cmdlet. The script must be on or accessible to your local computer. The results are returned to your local computer.

For example, the following command runs the `DiskCollect.ps1` script on the remote computers, Server01 and Server02.

```
PowerShell
```

```
Invoke-Command -ComputerName Server01, Server02 -FilePath  
C:\Scripts\DiskCollect.ps1
```

Establish a Persistent Connection

Use the [New-PSSession](#) cmdlet to create a persistent session on a remote computer. The following example creates remote sessions on Server01 and Server02. The session objects are stored in the `$s` variable.

```
PowerShell
```

```
$s = New-PSSession -ComputerName Server01, Server02
```

Now that the sessions are established, you can run any command in them. And because the sessions are persistent, you can collect data from one command and use it in another command.

For example, the following command runs a `Get-HotFix` command in the sessions in the `$s` variable and it saves the results in the `$h` variable. The `$h` variable is created in each of the sessions in `$s`, but it doesn't exist in the local session.

PowerShell

```
Invoke-Command -Session $s {$h = Get-HotFix}
```

Now you can use the data in the `$h` variable with other commands in the same session. The results are displayed on the local computer. For example:

PowerShell

```
Invoke-Command -Session $s {$h | where {$_.InstalledBy -ne "NT AUTHORITY\SYSTEM"}}
```

Advanced Remoting

PowerShell includes cmdlets that allow you to:

- Configure and create remote sessions both from the local and remote ends
- Create customized and restricted sessions
- Import commands from a remote session that actually run implicitly on the remote session
- Configure the security of a remote session

PowerShell on Windows includes a WSMAN provider. The provider creates a `WSMan:` drive that lets you navigate through a hierarchy of configuration settings on the local computer and remote computers.

For more information about the WSMAN provider, see [WSMAN Provider](#) and [About WS-Management Cmdlets](#), or in the Windows PowerShell console, type `Get-Help WSMAN`.

For more information, see:

- [PowerShell Remoting FAQ](#)

- [Register-PSSessionConfiguration](#)
- [Import-PSSession](#)

For help with remoting errors, see [about_Remote_Troubleshooting](#).

See Also

- [about_Remote](#)
- [about_Remote_Requirements](#)
- [about_Remote_Troubleshooting](#)
- [about_PSSessions](#)
- [about_WS-Management_Cmdlets](#)
- [Invoke-Command](#)
- [Import-PSSession](#)
- [New-PSSession](#)
- [Register-PSSessionConfiguration](#)
- [WSMan Provider](#)

PowerShell remoting over SSH

Article • 05/29/2025

Overview

PowerShell remoting normally uses WinRM for connection negotiation and data transport. SSH is now available for Linux and Windows platforms and allows true multiplatform PowerShell remoting.

WinRM provides a robust hosting model for PowerShell remote sessions. SSH-based remoting doesn't currently support remote endpoint configuration and Just Enough Administration (JEA).

SSH remoting lets you do basic PowerShell session remoting between Windows and Linux computers. SSH remoting creates a PowerShell host process on the target computer as an SSH subsystem. Eventually we'll implement a general hosting model, similar to WinRM, to support endpoint configuration and JEA.

The `New-PSSession`, `Enter-PSSession`, and `Invoke-Command` cmdlets now have a new parameter set to support this new remoting connection.

```
[ -HostName <string> ] [ -UserName <string> ] [ -KeyFilePath <string> ]
```

To create a remote session, you specify the target computer with the **HostName** parameter and provide the user name with **UserName**. When running the cmdlets interactively, you're prompted for a password. You can also use SSH key authentication using a private key file with the **KeyFilePath** parameter. Creating keys for SSH authentication varies by platform.

General setup information

PowerShell 6 or higher, and SSH must be installed on all computers. Install both the SSH client (`ssh.exe`) and server (`sshd.exe`) so that you can remote to and from the computers. OpenSSH for Windows is now available in Windows 10 build 1809 and Windows Server 2019. For more information, see [Manage Windows with OpenSSH](#). For Linux, install SSH, including `sshd` server, that's appropriate for your platform. You also need to install PowerShell from GitHub to get the SSH remoting feature. The SSH server must be configured to create an SSH subsystem to host a PowerShell process on the remote computer. And, you must enable **password** or **key-based** authentication.

Install the SSH service on a Windows computer

1. Install the latest version of PowerShell. For more information, see [Installing PowerShell on Windows](#).

You can confirm that PowerShell has SSH remoting support by listing the `New-PSSession` parameter sets. You'll notice there are parameter set names that begin with `SSH`. Those parameter sets include `SSH` parameters.

```
PowerShell
```

```
(Get-Command New-PSSession).ParameterSets.Name
```

```
Output
```

```
Name
```

```
----
```

```
SSHHost
```

```
SSHHostHashParam
```

2. Install the latest Win32 OpenSSH. For installation instructions, see [Getting started with OpenSSH](#).

 **Note**

If you want to set PowerShell as the default shell for OpenSSH, see [Configuring Windows for OpenSSH](#).

3. Edit the `sshd_config` file located at `$Env:ProgramData\ssh`.

- Make sure password authentication is enabled:

```
PasswordAuthentication yes
```

- Create the SSH subsystem that hosts a PowerShell process on the remote computer:

```
Subsystem powershell C:/program~1/powershell/7/pwsh.exe -sshs
```

Note

There is a bug in OpenSSH for Windows that prevents you from using a path with spaces for the subsystem executable. There are two ways to work around this issue:

- Use the Windows *8.3-style* short name for the PowerShell executable path
- Create a symbolic link to the PowerShell executable that results in a path without spaces

For more information, see [issue #784](#) in the PowerShell/Win32-OpenSSH repository.

You only need to get the 8.3-style name for the segment of the path that contains the space. By default PowerShell 7 is installed in `C:\Program Files\PowerShell\7\`. The 8.3-style name for `Program Files` should be `program~1`. You can use the following command to verify the name:

```
PowerShell
```

```
Get-CimInstance Win32_Directory -Filter 'Name="C:\\Program Files"' |  
    Select-Object EightDotThreeFileName
```

The 8.3 name is a legacy feature of the NTFS file system that can be disabled. This feature must be enabled for the volume on which PowerShell is installed.

Alternatively, you can create a symbolic link to the PowerShell executable that results in a path without spaces. This method is preferred because it allows you to update the link if the path to the PowerShell executable ever changes, without also needing to update your `sshd_config` file.

Use the following command to create a symbolic link to the executable:

```
PowerShell
```

```
$newItemSplat = @{  
    ItemType = 'SymbolicLink'  
    Path = 'C:\ProgramData\ssh\'  
    Name = 'pwsh.exe'  
    Value = (Get-Command pwsh.exe).Source  
}  
New-Item @newItemSplat
```

This command creates the symbolic link in the same directory used by the OpenSSH server to store the host keys and other configuration.

- Optionally, enable key authentication:

```
PubkeyAuthentication yes
```

For more information, see [Managing OpenSSH Keys](#).

4. Restart the `sshd` service.

```
PowerShell
```

```
Restart-Service sshd
```

5. Add the path where OpenSSH is installed to your PATH environment variable. For example, `C:\Program Files\OpenSSH\`. This entry allows for the `ssh.exe` to be found.

Install the SSH service on an Ubuntu Linux computer

1. Install the latest version of PowerShell, see [Installing PowerShell on Ubuntu](#).

2. Install [Ubuntu OpenSSH Server ↗](#).

```
Bash
```

```
sudo apt install openssh-client  
sudo apt install openssh-server
```

3. Edit the `sshd_config` file at location `/etc/ssh`.

- Make sure password authentication is enabled:

```
PasswordAuthentication yes
```

- Optionally, enable key authentication:

```
PubkeyAuthentication yes
```

For more information about creating SSH keys on Ubuntu, see the manpage for [ssh-keygen](#).

- Add a PowerShell subsystem entry:

```
Subsystem powershell /usr/bin/pwsh -sshs -NoLogo
```

! Note

The default location of the PowerShell executable is `/usr/bin/pwsh`. The location can vary depending on how you installed PowerShell.

4. Restart the ssh service.

Bash

```
sudo systemctl restart sshd.service
```

Install the SSH service on a macOS computer

1. Install the latest version of PowerShell. For more information, [Installing PowerShell on macOS](#).

Make sure SSH Remoting is enabled by following these steps:

- Open `System Settings`.
- Click on `General`.
- Click on `Sharing`.
- Check `Remote Login` to set `Remote Login: On`.
- Allow access to the appropriate users.

2. Edit the `sshd_config` file at location `/private/etc/ssh/sshd_config`.

Use a text editor such as `nano`:

Bash

```
sudo nano /private/etc/ssh/sshd_config
```

- Make sure password authentication is enabled:

```
PasswordAuthentication yes
```

- Add a PowerShell subsystem entry:

```
Subsystem powershell /usr/local/bin/pwsh -sshs -NoLogo
```

 **Note**

The default location of the PowerShell executable is `/usr/local/bin/pwsh`. The location can vary depending on how you installed PowerShell.

- Optionally, enable key authentication:

```
PubkeyAuthentication yes
```

3. Restart the `sshd` service.

Bash

```
sudo launchctl stop com.openssh.sshd
sudo launchctl start com.openssh.sshd
```

 **Note**

When you upgrade your operating system, the SSH configuration file might be overwritten. Make sure you check the configuration file after an upgrade.

Authentication

PowerShell remoting over SSH relies on the authentication exchange between the SSH client and SSH service and doesn't implement any authentication schemes itself. The result is that any configured authentication schemes including multi-factor authentication are handled by SSH and independent of PowerShell. For example, you can configure the SSH service to require public key authentication and a one-time password for added security. Configuration of multi-factor authentication is outside the scope of this documentation. Refer to documentation for SSH on how to correctly configure multi-factor authentication and validate it works outside of PowerShell before attempting to use it with PowerShell remoting.

! Note

Users retain the same privileges in remote sessions. Meaning, Administrators have access to an elevated shell, and normal users do not.

PowerShell remoting example

The easiest way to test remoting is to try it on a single computer. In this example, we create a remote session back to the same Linux computer. We're using PowerShell cmdlets interactively so we see prompts from SSH asking to verify the host computer and prompting for a password. You can do the same thing on a Windows computer to ensure remoting is working. Then, remote between computers by changing the host name.

Linux to Linux

PowerShell

```
$session = New-PSSession -HostName UbuntuVM1 -UserName TestUser
```

Output

```
The authenticity of host 'UbuntuVM1 (9.129.17.107)' can't be established.  
ECDSA key fingerprint is SHA256:2kCbnhT2dUE6WCggVJ8Hyfu1z2wE4lifaJXL07QJy0Y.  
Are you sure you want to continue connecting (yes/no)?  
TestUser@UbuntuVM1's password:
```

PowerShell

```
$session
```

Output

Id	Name	ComputerName	ComputerType	State	ConfigurationName
Availability					
--	--	--	--	--	--
1	SSH1	UbuntuVM1	RemoteMachine	Opened	DefaultShell
Available					

PowerShell

```
Enter-PSSession $session
```

Output

```
[UbuntuVM1]: PS /home/TestUser> uname -a
Linux TestUser-UbuntuVM1 4.2.0-42-generic 49~16.04.1-Ubuntu SMP Wed Jun 29
20:22:11 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
```

```
[UbuntuVM1]: PS /home/TestUser> Exit-PSSession
```

PowerShell

```
Invoke-Command $session -ScriptBlock { Get-Process pwsh }
```

Output

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
PSComputerName							
--	--	--	--	--	--	--	--
0	0	0	19	3.23	10635	635	pwsh
0	0	0	21	4.92	11033	017	pwsh
0	0	0	20	3.07	11076	076	pwsh

Linux to Windows

PowerShell

```
Enter-PSSession -HostName WinVM1 -UserName PTestName
```

PTestName@WinVM1's password:

PowerShell

```
[WinVM1]: PS C:\Users\PTestName\Documents> cmd /c ver
```

Output

```
Microsoft Windows [Version 10.0.10586]
```

Windows to Windows

PowerShell

```
C:\Users\PSUser\Documents>pwsh.exe
```

Output

```
PowerShell
Copyright (c) Microsoft Corporation. All rights reserved.
```

PowerShell

```
$session = New-PSSession -HostName WinVM2 -UserName PSRemoteUser
```

Output

```
The authenticity of host 'WinVM2 (10.13.37.3)' can't be established.
ECDSA key fingerprint is SHA256:kSU6s1AR0yQVMEynVIXAdxSiZpwDBigpAF/TXjjWjmw.
Are you sure you want to continue connecting (yes/no)?
Warning: Permanently added 'WinVM2,10.13.37.3' (ECDSA) to the list of known hosts.
PSRemoteUser@WinVM2's password:
```

PowerShell

```
$session
```

Output

Id	Name	ComputerName	ComputerType	State
ConfigurationName	Availability			
--	---	-----	-----	-----
-	-----			
1	SSH1	WinVM2	RemoteMachine	Opened
	Available			DefaultShell

PowerShell

```
Enter-PSSession -Session $session
```

Output

```
[WinVM2]: PS C:\Users\PSRemoteUser\Documents> $PSVersionTable
```

Name	Value
---	----
PSEdition	Core
PSCompatibleVersions	{1.0, 2.0, 3.0, 4.0...}
SerializationVersion	1.1.0.1
BuildVersion	3.0.0.0
CLRVersion	
PSVersion	6.0.0-alpha
WSManStackVersion	3.0
PSRemotingProtocolVersion	2.3
GitCommitId	v6.0.0-alpha.17

```
[WinVM2]: PS C:\Users\PSRemoteUser\Documents>
```

Limitations

- The `sudo` command doesn't work in a remote session to a Linux computer.
- PSRemoting over SSH doesn't support Profiles and doesn't have access to `$PROFILE`. Once in a session, you can load a profile by dot sourcing the profile with the full filepath. This isn't related to SSH profiles. You can configure the SSH server to use PowerShell as the default shell and to load a profile through SSH. See the SSH documentation for more information.
- Prior to PowerShell 7.1, remoting over SSH didn't support second-hop remote sessions. This capability was limited to sessions using WinRM. PowerShell 7.1 allows `Enter-PSSession` and `Enter-PSHostProcess` to work from within any interactive remote session.

See also

- [Installing PowerShell on Linux](#)
- [Installing PowerShell on macOS](#)
- [Installing PowerShell on Windows](#)
- [Manage Windows with OpenSSH](#)
- [Managing OpenSSH Keys](#)

- Ubuntu SSH ↗

Using WS-Management (WSMan) Remoting in PowerShell

Article • 01/27/2025

Enabling PowerShell remoting

To enable PowerShell remoting run the `Enable-PSRemoting` cmdlet in an elevated PowerShell session. Running `Enable-PSRemoting` configures a remoting endpoint for the specific installation version that you are running the cmdlet in. For example, when you run `Enable-PSRemoting` while running PowerShell 7.4, PowerShell creates a remoting endpoint runs PowerShell 7.4. If you run `Enable-PSRemoting` while running PowerShell 7-preview, PowerShell creates a remoting endpoint that runs PowerShell 7-preview. You can create multiple remoting endpoints for different versions of that run side-by-side.

Running `Enable-PSRemoting` creates two endpoints for that version.

- One has a simple name corresponding to the PowerShell major version. that hosts the session. For example, **PowerShell.7.4**.
- The other configuration name contains the full version number. For example, **PowerShell.7.4.7**.

You can connect to the latest version of PowerShell 7 host version using the simple name, **PowerShell.7.4**. You can connect to a specific version of PowerShell using the longer, version-specific name.

Use the **ConfigurationName** parameter with the `New-PSSession` and `Enter-PSSession` cmdlets to connect to a named configuration.

Remoting to older versions of Windows

The following prerequisites must be met to enable PowerShell remoting over WSMan on older versions of Windows.

- Install the Windows Management Framework (WMF) 5.1 (as necessary). For more information about WMF, see [WMF Overview](#).
- Install the [Universal C Runtime](#) on Windows versions predating Windows 10. It's available via direct download or Windows Update. Fully patched systems already have this package installed.

WSMan remoting isn't supported on non-Windows platforms

Since the release of PowerShell 6, support for remoting over WS-Management (WSMan) on non-Windows platforms has only been available to a limited set of Linux distributions. All versions of those distributions that supported WSMan are no longer supported by the Linux vendors that created them.

On non-Windows, WSMan relied on the [Open Management Infrastructure \(OMI\)](#) project, which no longer supports PowerShell remoting. The OMI WSMan client is dependent on [OpenSSL 1.0](#). Most Linux distributions have moved to [OpenSSL 2.0](#), which isn't backward-compatible. At this time, there is no supported distribution that has the dependencies needed for the OMI WSMan client to work.

The outdated libraries and supporting code have been removed for non-Windows platforms. WSMan-based remoting is still supported between Windows systems. Remoting over SSH is supported for all platforms. For more information, see [PowerShell remoting over SSH](#).

ⓘ Note

Users may be able to get WSMan remoting to work using the [PSWSMan](#) module. This module isn't supported or maintained by Microsoft.

Further reading

- [Enable-PSRemoting](#)
- [Enter-PSSession](#)
- [New-PSSession](#)

Security Considerations for PowerShell Remoting using WinRM

Article • 03/24/2025

PowerShell Remoting is the recommended way to manage Windows systems. PowerShell Remoting is enabled by default in Windows Server 2012 R2 and higher. This document covers security concerns, recommendations, and best practices when using PowerShell Remoting.

What is PowerShell Remoting?

PowerShell Remoting uses [Windows Remote Management \(WinRM\)](#) to allow users to run PowerShell commands on remote computers. WinRM is the Microsoft implementation of the [Web Services for Management \(WS-Management\)](#) ↗ protocol. You can find more information about using PowerShell Remoting at [Running Remote Commands](#).

PowerShell Remoting isn't the same as using the `ComputerName` parameter of a cmdlet to run it on a remote computer, which uses Remote Procedure Call (RPC) as its underlying protocol.

PowerShell Remoting default settings

PowerShell Remoting with WinRM listens on the following ports:

- HTTP: 5985
- HTTPS: 5986

By default, PowerShell Remoting only allows connections from members of the Administrators group. Sessions are launched under the user's context, so all operating system access controls applied to individual users and groups continue to apply to them while connected over PowerShell Remoting.

On private networks, the default Windows Firewall rule for PowerShell Remoting accepts all connections. On public networks, the default Windows Firewall rule allows PowerShell Remoting connections only from within the same subnet. You have to explicitly change that rule to open PowerShell Remoting to all connections on a public network.

Warning

The firewall rule for public networks is meant to protect the computer from potentially malicious external connection attempts. Use caution when removing this rule.

Process isolation

PowerShell Remoting uses WinRM for communication between computers. WinRM runs as a service under the Network Service account, and spawns isolated processes running as user accounts to host PowerShell instances. An instance of PowerShell running as one user has no access to a process running an instance of PowerShell as another user.

Event logs generated by PowerShell Remoting

Researchers from Mandiant presented a session at the BlackHat conference that provides a good summary of the event logs and other security evidence generated by PowerShell Remoting sessions. For more information, see [Investigating PowerShell Attacks ↗](#).

Encryption and transport protocols

It's helpful to consider the security of a PowerShell Remoting connection from two perspectives: initial authentication, and ongoing communication.

Regardless of the transport protocol used (HTTP or HTTPS), WinRM always encrypts all PowerShell remoting communication after initial authentication.

Initial authentication

Authentication confirms the identity of the client to the server - and ideally - the server to the client.

When a client connects to a domain server using its computer name, the default authentication protocol is [Kerberos](#). Kerberos guarantees both the user identity and server identity without sending any sort of reusable credential.

When a client connects to a domain server using its IP address, or connects to a workgroup server, Kerberos authentication isn't possible. In that case, PowerShell Remoting relies on the [NTLM authentication protocol](#). The NTLM authentication protocol guarantees the user identity without sending any sort of delegable credential. To prove user identity, the NTLM protocol requires that both the client and server

compute a session key from the user's password without ever exchanging the password itself. The server typically doesn't know the user's password, so it communicates with the domain controller, which does know the user's password and calculates the session key for the server.

The NTLM protocol doesn't, however, guarantee server identity. As with all protocols that use NTLM for authentication, an attacker with access to a domain-joined computer's machine account could invoke the domain controller to compute an NTLM session-key and impersonate the server.

NTLM-based authentication is disabled by default. You can enable NTLM by either configuring SSL on the target server, or by configuring the WinRM TrustedHosts setting on the client.

Using SSL certificates to validate server identity during NTLM-based connections

Since the NTLM authentication protocol can't ensure the identity of the target server (only that it already knows your password), you can configure target servers to use SSL for PowerShell Remoting. Assigning an SSL certificate to the target server (if issued by a Certificate Authority that the client also trusts) enables NTLM-based authentication that guarantees both the user identity and server identity.

Ignoring NTLM-based server identity errors

If deploying an SSL certificate to a server for NTLM connections is infeasible, you can suppress the resulting identity errors by adding the server to the WinRM **TrustedHosts** list. Adding a server name to the **TrustedHosts** list shouldn't be considered as any form of statement of the trustworthiness of the hosts themselves - as the NTLM authentication protocol can't guarantee that you are in fact connecting to the host you're intending to connect to. Instead, you should consider the **TrustedHosts** setting to be the list of hosts for which you wish to suppress the error generated by being unable to verify the server's identity.

Ongoing Communication

Once initial authentication is complete, the WinRM encrypts the ongoing communication. When you connect over HTTPS, WinRM uses the TLS protocol to negotiate the encryption used to transport data. When you connect over HTTP, WinRM uses the message-level encryption negotiated by the initial authentication protocol.

- Basic authentication provides no encryption.
- NTLM authentication uses an RC4 cipher with a 128-bit key.
- The `etype` in the TGS ticket determines Kerberos authentication encryption. This is AES-256 on modern systems.
- CredSSP encryption uses the TLS cipher suite negotiated in the handshake.

Making the second hop

By default, PowerShell Remoting uses Kerberos (if available) or NTLM for authentication. Both of these protocols authenticate to the remote machine without sending credentials to it. This is the most secure way to authenticate, but because the remote machine doesn't have the user's credentials, it can't access other computers and services on the user's behalf. This is known as the *second hop problem*.

There are several ways to avoid this problem. For descriptions of these methods, and the pros and cons of each, see [Making the second hop in PowerShell Remoting](#).

References

- [Windows Remote Management \(WinRM\)](#)
- [Web Services for Management \(WS-Management\) ↗](#)
- [2.2.9.1 Encrypted Message Types](#)
- [Kerberos](#)
- [NTLM authentication protocol](#)
- [Investigating PowerShell Attacks ↗](#)

Making the second hop in PowerShell Remoting

Article • 04/01/2024

The "second hop problem" refers to a situation like the following:

1. You are logged in to *ServerA*.
2. From *ServerA*, you start a remote PowerShell session to connect to *ServerB*.
3. A command you run on *ServerB* via your PowerShell Remoting session attempts to access a resource on *ServerC*.
4. Access to the resource on *ServerC* is denied, because the credentials you used to create the PowerShell Remoting session aren't passed from *ServerB* to *ServerC*.

There are several ways to address this problem. The following table lists the methods in order of preference.

[+] Expand table

Configuration	Note
CredSSP	Balances ease of use and security
Resource-based Kerberos constrained delegation	Higher security with simpler configuration
Kerberos constrained delegation	High security but requires Domain Administrator
Kerberos delegation (unconstrained)	Not recommended
Just Enough Administration (JEA)	Can provide the best security but requires more detailed configuration
PSSessionConfiguration using RunAs	Simpler to configure but requires credential management
Pass credentials inside an <code>Invoke-Command</code> script block	Simplest to use but you must provide credentials

CredSSP

You can use the [Credential Security Support Provider \(CredSSP\)](#) for authentication.

CredSSP caches credentials on the remote server (*ServerB*), so using it opens you up to credential theft attacks. If the remote computer is compromised, the attacker has access to the user's credentials. CredSSP is disabled by default on both client and server

computers. You should enable CredSSP only in the most trusted environments. For example, a domain administrator connecting to a domain controller because the domain controller is highly trusted.

For more information about security concerns when using CredSSP for PowerShell Remoting, see [Accidental Sabotage: Beware of CredSSP ↴](#).

For more information about credential theft attacks, see [Mitigating Pass-the-Hash \(PtH\) Attacks and Other Credential Theft ↴](#).

For an example of how to enable and use CredSSP for PowerShell remoting, see [Enable PowerShell "Second-Hop" Functionality with CredSSP ↴](#).

Pros

- It works for all servers with Windows Server 2008 or later.

Cons

- Has security vulnerabilities.
- Requires configuration of both client and server roles.
- doesn't work with the Protected Users group. For more information, see [Protected Users Security Group](#).

Kerberos constrained delegation

You can use legacy constrained delegation (not resource-based) to make the second hop. Configure Kerberos constrained delegation with the option "Use any authentication protocol" to allow protocol transition.

Pros

- Requires no special coding
- Credentials aren't stored.

Cons

- Doesn't support the second hop for WinRM.
- Requires Domain Administrator access to configure.
- Must be configured on the Active Directory object of the remote server (*ServerB*).
- Limited to one domain. Can't cross domains or forests.

- Requires rights to update objects and Service Principal Names (SPNs).
- *ServerB* can acquire a Kerberos ticket to *ServerC* on behalf of the user without user intervention.

 Note

Active Directory accounts that have the **Account is sensitive and can't be delegated** property set can't be delegated. For more information, see [Security Focus: Analysing 'Account is sensitive and can't be delegated' for Privileged Accounts](#) and [Kerberos Authentication Tools and Settings](#).

Resource-based Kerberos constrained delegation

Using resource-based Kerberos constrained delegation (introduced in Windows Server 2012), you configure credential delegation on the server object where resources reside. In the second hop scenario described above, you configure *ServerC* to specify from where it accepts delegated credentials.

Pros

- Credentials aren't stored.
- Configured using PowerShell cmdlets. No special coding required.
- Doesn't require Domain Administrator access to configure.
- Works across domains and forests.

Cons

- Requires Windows Server 2012 or later.
- Doesn't support the second hop for WinRM.
- Requires rights to update objects and Service Principal Names (SPNs).

 Note

Active Directory accounts that have the **Account is sensitive and can't be delegated** property set can't be delegated. For more information, see [Security Focus: Analysing 'Account is sensitive and can't be delegated' for Privileged Accounts](#) and [Kerberos Authentication Tools and Settings](#).

Example

Let's look at a PowerShell example that configures resource-based constrained delegation on *ServerC* to allow delegated credentials from a *ServerB*. This example assumes that all servers are running supported versions of Windows Server, and that there is at least one Windows domain controller for each trusted domain.

Before you can configure constrained delegation, you must add the `RSAT-AD-PowerShell` feature to install the Active Directory PowerShell module, and then import that module into your session:

```
PowerShell

Add-WindowsFeature RSAT-AD-PowerShell
Import-Module ActiveDirectory
Get-Command -ParameterName PrincipalsAllowedToDelegateToAccount
```

Several available cmdlets now have a `PrincipalsAllowedToDelegateToAccount` parameter:

Output		
CommandType	Name	ModuleName
Cmdlet	New-ADComputer	ActiveDirectory
Cmdlet	New-ADServiceAccount	ActiveDirectory
Cmdlet	New-ADUser	ActiveDirectory
Cmdlet	Set-ADComputer	ActiveDirectory
Cmdlet	Set-ADServiceAccount	ActiveDirectory
Cmdlet	Set-ADUser	ActiveDirectory

The `PrincipalsAllowedToDelegateToAccount` parameter sets the Active Directory object attribute `msDS-AllowedToActOnBehalfOfOtherIdentity`, which contains an access control list (ACL) that specifies which accounts have permission to delegate credentials to the associated account (in our example, it will be the machine account for *ServerA*).

Now let's set up the variables we'll use to represent the servers:

```
PowerShell

# Set up variables for reuse
$ServerA = $Env:COMPUTERNAME
$ServerB = Get-ADComputer -Identity ServerB
$ServerC = Get-ADComputer -Identity ServerC
```

WinRM (and therefore PowerShell remoting) runs as the computer account by default. You can see this by looking at the **StartName** property of the `winrm` service:

PowerShell

```
Get-CimInstance Win32_Service -Filter 'Name="winrm"' | Select-Object StartName
```

Output

```
StartName  
-----  
NT AUTHORITY\NetworkService
```

For *ServerC* to allow delegation from a PowerShell remoting session on *ServerB*, we must set the **PrincipalsAllowedToDelegateToAccount** parameter on *ServerC* to the computer object of *ServerB*:

PowerShell

```
# Grant resource-based Kerberos constrained delegation
Set-ADComputer -Identity $ServerC -PrincipalsAllowedToDelegateToAccount
$ServerB

# Check the value of the attribute directly
$x = Get-ADComputer -Identity $ServerC -Properties msDS-
AllowedToActOnBehalfOfOtherIdentity
$x.'msDS-AllowedToActOnBehalfOfOtherIdentity'.Access

# Check the value of the attribute indirectly
Get-ADComputer -Identity $ServerC -Properties
PrincipalsAllowedToDelegateToAccount
```

The Kerberos [Key Distribution Center \(KDC\)](#) caches denied-access attempts (negative cache) for 15 minutes. If *ServerB* has previously attempted to access *ServerC*, you need to clear the cache on *ServerB* by invoking the following command:

PowerShell

```
Invoke-Command -ComputerName $ServerB.Name -Credential $cred -ScriptBlock {
    klist purge -li 0x3e7
}
```

You could also restart the computer, or wait at least 15 minutes to clear the cache.

After clearing the cache, you can successfully run code from *ServerA* through *ServerB* to *ServerC*:

```
PowerShell

# Capture a credential
$cred = Get-Credential Contoso\Alice

# Test kerberos double hop
Invoke-Command -ComputerName $ServerB.Name -Credential $cred -ScriptBlock {
    Test-Path \\$($Using:ServerC.Name)\C$
    Get-Process lsass -ComputerName $($Using:ServerC.Name)
    Get-EventLog -LogName System -Newest 3 -ComputerName
    $($Using:ServerC.Name)
}
```

In this example, the `Using:` scope modifier is used to make the `$ServerC` variable visible to *ServerB*. For more information about the `Using:` scope modifier, see [about_Remote_Variables](#).

To allow multiple servers to delegate credentials to *ServerC*, set the value of the `PrincipalsAllowedToDelegateToAccount` parameter on *ServerC* to an array:

```
PowerShell

# Set up variables for each server
$ServerB1 = Get-ADComputer -Identity ServerB1
$ServerB2 = Get-ADComputer -Identity ServerB2
$ServerB3 = Get-ADComputer -Identity ServerB3
$ServerC = Get-ADComputer -Identity ServerC

$servers = @(
    $ServerB1,
    $ServerB2,
    $ServerB3
)

# Grant resource-based Kerberos constrained delegation
Set-ADComputer -Identity $ServerC -PrincipalsAllowedToDelegateToAccount
$servers
```

If you want to make the second hop across domains, use the `Server` parameter to specify fully-qualified domain name (FQDN) of the domain controller of the domain to which *ServerB* belongs:

```
PowerShell
```

```
# For ServerC in Contoso domain and ServerB in other domain
$ServerB = Get-ADComputer -Identity ServerB -Server dc1.alpineskihouse.com
$ServerC = Get-ADComputer -Identity ServerC
Set-ADComputer -Identity $ServerC -PrincipalsAllowedToDelegateToAccount
$ServerB
```

To remove the ability to delegate credentials to ServerC, set the value of the **PrincipalsAllowedToDelegateToAccount** parameter on *ServerC* to `$null`:

PowerShell

```
Set-ADComputer -Identity $ServerC -PrincipalsAllowedToDelegateToAccount
$null
```

Information on resource-based Kerberos constrained delegation

- [What's New in Kerberos Authentication](#)
- [How Windows Server 2012 Eases the Pain of Kerberos Constrained Delegation, Part 1 ↗](#)
- [How Windows Server 2012 Eases the Pain of Kerberos Constrained Delegation, Part 2 ↗](#)
- [Understanding Kerberos Constrained Delegation for Microsoft Entra application proxy deployments with Integrated Windows Authentication ↗](#)
- [\[MS-ADA2\] Active Directory Schema Attributes M2.210 Attribute msDS-AllowedToActOnBehalfOfOtherIdentity\]MS-ADA2](#)
- [\[MS-SFU\] Kerberos Protocol Extensions: Service for User and Constrained Delegation Protocol 1.3.2 S4U2proxy\]MS-SFU](#)
- [Remote Administration Without Constrained Delegation Using PrincipalsAllowedToDelegateToAccount](#)

Kerberos delegation (unconstrained)

You can also use Kerberos unconstrained delegation to make the second hop. Like all Kerberos scenarios, credentials aren't stored. This method doesn't support the second hop for WinRM.

 **Warning**

This method provides no control of where delegated credentials are used. It's less secure than CredSSP. This method should only be used for testing scenarios.

Just Enough Administration (JEA)

JEA allows you to restrict what commands an administrator can run during a PowerShell session. It can be used to solve the second hop problem.

For information about JEA, see [Just Enough Administration](#).

Pros

- No password maintenance when using a virtual account.

Cons

- Requires WMF 5.0 or later.
- Requires configuration on every intermediate server (*ServerB*).

PSSessionConfiguration using RunAs

You can create a session configuration on *ServerB* and set its **RunAsCredential** parameter.

For information about using **PSSessionConfiguration** and **RunAs** to solve the second hop problem, see [Another solution to multi-hop PowerShell remoting](#).

Pros

- Works with any server with WMF 3.0 or later.

Cons

- Requires configuration of **PSSessionConfiguration** and **RunAs** on every intermediate server (*ServerB*).
- Requires password maintenance when using a domain **RunAs** account

Pass credentials inside an Invoke-Command script block

You can pass credentials inside the `ScriptBlock` parameter of a call to the [Invoke-Command](#) cmdlet.

Pros

- Doesn't require special server configuration.
- Works on any server running WMF 2.0 or later.

Cons

- Requires an awkward code technique.
- If running WMF 2.0, requires different syntax for passing arguments to a remote session.

Example

The following example shows how to pass credentials in a script block:

PowerShell

```
# This works without delegation, passing fresh creds
# Note $Using:Cred in nested request
$cred = Get-Credential Contoso\Administrator
Invoke-Command -ComputerName ServerB -Credential $cred -ScriptBlock {
    hostname
    Invoke-Command -ComputerName ServerC -Credential $Using:cred -
    ScriptBlock {hostname}
}
```

See also

[PowerShell Remoting Security Considerations](#)

Securing a restricted PowerShell remoting session

Article • 04/01/2024

There are scenarios where you want to host a PowerShell session that, for security reasons, has been limited to a subset of PowerShell commands.

By definition, a restricted session is one where `Import-Module` isn't allowed to be used. There may be other limitations, but this is the primary requirement. If the user can import a module, then they can run anything they want.

Examples of restricted sessions include:

- Just-Enough-Administration (JEA)
- Custom restricted remoting implementations such as the Exchange and Teams modules

For most system administrators, JEA provides the best experience for creating restricted sessions and should be your first choice. For more information about JEA, see the [JEA Overview](#).

Recommendations for custom session implementations

If your scenario requires a custom implementation, then you should follow these recommendations.

Limit the use and capabilities of PowerShell providers

Review how the allowed providers are used to ensure that you don't create vulnerabilities in your restricted session implementation.

Warning

Don't allow the `FileSystem` provider. If users can write to any part of the file system, it's possible to completely bypass security.

Don't allow the `Certificate` provider. With the provider enabled, a user could gain access to stored private keys.

Don't allow commands that can create new runspaces

⚠ Warning

The `*-Job` cmdlets can create new runspaces without the restrictions.

Don't allow the `Trace-Command` cmdlet.

⚠ Warning

Using `Trace-Command` brings all traced commands into the session.

Don't create your own proxy implementations for the restricted commands

PowerShell has a set of proxy commands for restricted command scenarios. These proxy commands ensure that input parameters can't compromise the security of the session.

The following commands have restricted proxies:

- `Exit-PSSession`
- `Get-Command`
- `Get-FormatData`
- `Get-Help`
- `Measure-Object`
- `Out-Default`
- `Select-Object`

If you create your own implementation of these commands, you may inadvertently allow users to run code prohibited by the JEA proxy commands.

You can run the following command to get a list of restricted commands:

PowerShell

```
$commands =
[System.Management.Automation.CommandMetadata]::GetRestrictedCommands(
    [System.Management.Automation.SessionCapabilities]::RemoteServer
)
```

You can examine the restricted proxy commands by using the following command:

PowerShell

```
$commands =  
[System.Management.Automation.CommandMetadata]::GetRestrictedCommands(  
    [System.Management.Automation.SessionCapabilities]::RemoteServer  
)  
$getHelpProxyBlock =  
[System.Management.Automation.ProxyCommand]::Create($commands['Get-Help'])
```

Configure the session to use NoLanguage mode

PowerShell `NoLanguage` mode disables the PowerShell scripting language completely. You can't run scripts or use variables. You can only run native commands and cmdlets.

For more information about language modes, see [about_Language_Modes](#).

Don't allow the debugger to be used in the session

By default, the PowerShell debugger runs code in `FullLanguage` mode. Set the `UseFullLanguageModeInDebugger` property in the `SessionState` to false.

For more information, see [UseFullLanguageModeInDebugger](#).



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

PowerShell Remoting FAQ

FAQ

When you work remotely, you type commands in PowerShell on one computer (known as the "local computer"), but the commands run on another computer (known as the "remote computer"). The experience of working remotely should be as much like working directly at the remote computer as possible.

Note

To use PowerShell remoting, the remote computer must be configured for remoting. For more information, see [about_Remote_Requirements](#).

Must both computers have PowerShell installed?

Yes. To work remotely, the local and remote computers must have PowerShell, the Microsoft .NET Framework, and the Web Services for Management (WS-Management) protocol. Any files and other resources that are needed to execute a particular command must be on the remote computer.

Computers running Windows PowerShell 3.0 and computers running Windows PowerShell 2.0 can connect to each other remotely and run remote commands. However, some features, such as the ability to disconnect from a session and reconnect to it, work only when both computers are running Windows PowerShell 3.0.

You must have permission to connect to the remote computer, permission to run PowerShell, and permission to access data stores (such as files and folders), and the registry on the remote computer.

For more information, see [about_Remote_Requirements](#).

How does remoting work?

When you submit a remote command, the command is transmitted across the network to the PowerShell engine on the remote computer, and it runs in the PowerShell client on the remote computer. The command results are sent back to the local computer and appear in the PowerShell session on the local computer.

To transmit the commands and receive the output, PowerShell uses the WS-Management protocol. For information about the WS-Management protocol, see [WS-Management Protocol](#) in the Windows documentation.

Beginning in Windows PowerShell 3.0, remote sessions are stored on the remote computer. This enables you to disconnect from the session and reconnect from a different session or a different computer without interrupting the commands or losing state.

Is PowerShell remoting secure?

When you connect to a remote computer, the system uses the username and password credentials on the local computer or the credentials that you supply in the command to log you in to the remote computer. The credentials and the rest of the transmission are encrypted.

To add additional protection, you can configure the remote computer to use Secure Sockets Layer (SSL) instead of HTTP to listen for Windows Remote Management (WinRM) requests. Then, users can use the `UseSSL` parameter of the `Invoke-Command`, `New-PSSession`, and `Enter-PSSession` cmdlets when establishing a connection. This option uses the more secure HTTPS channel instead of HTTP.

Do all remote commands require PowerShell remoting?

No. Some cmdlets have a `ComputerName` parameter that lets you get objects from the remote computer.

These cmdlets do not use PowerShell remoting. So, you can use them on any computer that is running PowerShell, even if the computer is not configured for PowerShell remoting or if the computer does not meet the requirements for PowerShell remoting.

These cmdlets include the following:

- `Get-HotFix`
- `Rename-Computer`
- `Restart-Computer`
- `Stop-Computer`

To find all the cmdlets with a `ComputerName` parameter, type:

```
PowerShell
```

```
Get-Help * -Parameter ComputerName  
# or  
Get-Command -ParameterName ComputerName
```

To determine whether the **ComputerName** parameter of a particular cmdlet requires PowerShell remoting, see the parameter description. To display the parameter description, type:

```
PowerShell
```

```
Get-Help <cmdlet-name> -Parameter ComputerName
```

For example:

```
PowerShell
```

```
Get-Help Get-HotFix -Parameter ComputerName
```

For all other commands, use the `Invoke-Command` cmdlet.

How do I run a command on a remote computer?

To run a command on a remote computer, use the `Invoke-Command` cmdlet.

Enclose your command in braces (`{}`) to make it a script block. Use the **ScriptBlock** parameter of `Invoke-Command` to specify the command.

You can use the **ComputerName** parameter of `Invoke-Command` to specify a remote computer. Or, you can create a persistent connection to a remote computer (a session) and then use the **Session** parameter of `Invoke-Command` to run the command in the session.

For example, the following commands run a `Get-Process` command remotely.

```
PowerShell
```

```
Invoke-Command -ComputerName Server01, Server02 -ScriptBlock {Get-Process}  
# - OR -
```

```
Invoke-Command -Session $s -ScriptBlock {Get-Process}
```

To interrupt a remote command, type `CTRL + C`. The interruption request is passed to the remote computer, where it terminates the remote command.

For more information about remote commands, see [about_Remote](#) and the Help topics for the cmdlets that support remoting.

Can I just telnet into a remote computer?

You can use the `Enter-PSSession` cmdlet to start an interactive session with a remote computer.

At the PowerShell prompt, type:

```
PowerShell  
Enter-PSSession <ComputerName>
```

The command prompt changes to show that you are connected to the remote computer.

```
<ComputerName>\C:>
```

Now, the commands that you type run on the remote computer just as though you typed them directly on the remote computer.

To end the interactive session, type:

```
PowerShell  
Exit-PSSession
```

An interactive session is a persistent session that uses the WS-Management protocol. It is not the same as using Telnet, but it provides a similar experience.

For more information, see [Enter-PSSession](#).

Can I create a persistent connection?

Yes. You can run remote commands by specifying the name of the remote computer, its NetBIOS name, or its IP address. Or, you can run remote commands by specifying a PowerShell session (PSSession) that is connected to the remote computer.

When you use the **ComputerName** parameter of `Invoke-Command` or `Enter-PSSession`, PowerShell establishes a temporary connection. PowerShell uses the connection to run only the current command, and then it closes the connection. This is a very efficient method for running a single command or several unrelated commands, even on many remote computers.

When you use the `New-PSSession` cmdlet to create a PSSession, PowerShell establishes a persistent connection for the PSSession. Then, you can run multiple commands in the PSSession, including commands that share data.

Typically, you create a PSSession to run a series of related commands that share data. Otherwise, the temporary connection created by the **ComputerName** parameter is sufficient for most commands.

For more information about sessions, see [about_PSSessions](#).

Can I run commands on more than one computer at a time?

Yes. The **ComputerName** parameter of the `Invoke-Command` cmdlet accepts multiple computer names, and the **Session** parameter accepts multiple PSSessions.

When you run an `Invoke-Command` command, PowerShell runs the commands on all of the specified computers or in all of the specified PSSessions.

PowerShell can manage hundreds of concurrent remote connections. However, the number of remote commands that you can send might be limited by the resources of your computer and its capacity to establish and maintain multiple network connections.

For more information, see the example in the `Invoke-Command` Help topic.

Where are my profiles?

PowerShell profiles are not run automatically in remote sessions, so the commands that the profile adds are not present in the session. In addition, the `$PROFILE` automatic

variable is not populated in remote sessions.

To run a profile in a session, use the `Invoke-Command` cmdlet.

For example, the following command runs the `CurrentUserCurrentHost` profile from the local computer in the session in `$s`.

```
Invoke-Command -Session $s -FilePath $PROFILE
```

The following command runs the `CurrentUserCurrentHost` profile from the remote computer in the session in `$s`. Because the `$PROFILE` variable is not populated, the command uses the explicit path to the profile.

PowerShell

```
Invoke-Command -Session $s {  
    . "$HOME\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1"  
}
```

After running this command, the commands that the profile adds to the session are available in `$s`.

You can also use a startup script in a session configuration to run a profile in every remote session that uses the session configuration.

For more information about PowerShell profiles, see [about_Profiles](#). For more information about session configurations, see [Register-PSSessionConfiguration](#).

How does throttling work on remote commands?

To help you manage the resources on your local computer, PowerShell includes a per-command throttling feature that lets you limit the number of concurrent remote connections that are established for each command.

The default is 32 concurrent connections, but you can use the `ThrottleLimit` parameter of the cmdlets to set a custom throttle limit for particular commands.

When you use the throttling feature, remember that it is applied to each command, not to the entire session or to the computer. If you are running commands concurrently in

several sessions or PSSessions, the number of concurrent connections is the sum of the concurrent connections in all the sessions.

To find cmdlets with a **ThrottleLimit** parameter, type:

```
Get-Help * -Parameter ThrottleLimit  
-or-  
Get-Command -ParameterName ThrottleLimit
```

Is the output of remote commands different from local output?

When you use PowerShell locally, you send and receive "live" .NET Framework objects; "live" objects are objects that are associated with actual programs or system components. When you invoke the methods or change the properties of live objects, the changes affect the actual program or component. And, when the properties of a program or component change, the properties of the object that represent them also change.

However, because most live objects cannot be transmitted over the network, PowerShell "serializes" most of the objects sent in remote commands, that is, it converts each object into a series of XML (Constraint Language in XML [CLiXML]) data elements for transmission.

When PowerShell receives a serialized object, it converts the XML into a deserialized object type. The deserialized object is an accurate record of the properties of the program or component at a previous time, but it is no longer "live", that is, it is no longer directly associated with the component. And, the methods are removed because they are no longer effective.

Typically, you can use deserialized objects just as you would use live objects, but you must be aware of their limitations. Also, the objects that are returned by the `Invoke-Command` cmdlet have additional properties that help you to determine the origin of the command.

Some object types, such as DirectoryInfo objects and GUIDs, are converted back into live objects when they are received. These objects do not need any special handling or formatting.

For information about interpreting and formatting remote output, see [about_Remote_Output](#).

Can I run background jobs remotely?

Yes. A PowerShell background job is a PowerShell command that runs asynchronously without interacting with the session. When you start a background job, the command prompt returns immediately, and you can continue to work in the session while the job runs even if it runs for an extended period of time.

You can start a background job even while other commands are running because background jobs always run asynchronously in a temporary session.

You can run background jobs on a local or remote computer. By default, a background job runs on the local computer. However, you can use the **AsJob** parameter of the `Invoke-Command` cmdlet to run any remote command as a background job. And, you can use `Invoke-Command` to run a `Start-Job` command remotely.

For more information about background jobs in PowerShell , see [about_Jobs](#) and [about_Remote_Jobs](#).

Can I run Windows programs on a remote computer?

You can use PowerShell remote commands to run Windows-based programs on remote computers. For example, you can run `Shutdown.exe` or `Ipconfig.exe` on a remote computer.

However, you cannot use PowerShell commands to open the user interface for any program on a remote computer.

When you start a Windows program on a remote computer, the command is not completed, and the PowerShell command prompt does not return, until the program is finished or until you press `CTRL + C` to interrupt the command. For example, if you run the `Ipconfig.exe` program on a remote computer, the command prompt does not return until `Ipconfig.exe` is completed.

If you use remote commands to start a program that has a user interface, the program process starts, but the user interface does not appear. The PowerShell command is not completed, and the command prompt does not return until you stop the program

process or until you press `CTRL + C`, which interrupts the command and stops the process.

For example, if you use a PowerShell command to run `Notepad` on a remote computer, the Notepad process starts on the remote computer, but the Notepad user interface does not appear. To interrupt the command and restore the command prompt, press `CTRL + C`.

Can I limit the commands that users can run remotely on my computer?

Yes. Every remote session must use one of the session configurations on the remote computer. You can manage the session configurations on your computer (and the permissions to those session configurations) to determine who can run commands remotely on your computer and which commands they can run.

A session configuration configures the environment for the session. You can define the configuration by using an assembly that implements a new configuration class or by using a script that runs in the session. The configuration can determine the commands that are available in the session. And, the configuration can include settings that protect the computer, such as settings that limit the amount of data that the session can receive remotely in a single object or command. You can also specify a security descriptor that determines the permissions that are required to use the configuration.

The `Enable-PSRemoting` cmdlet creates the default session configurations on your computer: `Microsoft.PowerShell`, `Microsoft.PowerShell.Workflow`, and `Microsoft.PowerShell32` (64-bit operating systems only). `Enable-PSRemoting` sets the security descriptor for the configuration to allow only members of the Administrators group on your computer to use them.

You can use the session configuration cmdlets to edit the default session configurations, to create new session configurations, and to change the security descriptors of all the session configurations.

Beginning in Windows PowerShell 3.0, the `New-PSSessionConfigurationFile` cmdlet lets you create custom session configurations by using a text file. The file includes options for setting the language mode and for specifying the cmdlets and modules that are available in sessions that use the session configuration.

When users use the `Invoke-Command`, `New-PSSession`, or `Enter-PSSession` cmdlets, they can use the `ConfigurationName` parameter to indicate the session configuration that is

used for the session. And, they can change the default configuration that their sessions use by changing the value of the `$PSSessionConfigurationName` preference variable in the session.

For more information about session configurations, see the Help for the session configuration cmdlets. To find the session configuration cmdlets, type:

```
PowerShell
```

```
Get-Command *PSSessionConfiguration
```

What are fan in and fan out configurations?

The most common PowerShell remoting scenario involving multiple computers is the one-to-many configuration, in which one local computer (the administrator's computer) runs PowerShell commands on numerous remote computers. This is known as the "fan-out" scenario.

However, in some enterprises, the configuration is many-to-one, where many client computers connect to a single remote computer that is running PowerShell, such as a file server or a kiosk. This is known as the "fan-in" configuration.

PowerShell remoting supports both fan-out and fan-in configurations.

For the fan-out configuration, PowerShell uses the Web Services for Management (WS-Management) protocol and the WinRM service that supports the Microsoft implementation of WS-Management. When a local computer connects to a remote computer, WS-Management establishes a connection and uses a plug-in for PowerShell to start the PowerShell host process (`Wsmprovhost.exe`) on the remote computer. The user can specify an alternate port, an alternate session configuration, and other features to customize the remote connection.

To support the "fan-in" configuration, PowerShell uses internet Information Services (IIS) to host WS-Management, to load the PowerShell plug-in, and to start PowerShell. In this scenario, instead of starting each PowerShell session in a separate process, all PowerShell sessions run in the same host process.

IIS hosting and fan-in remote management is not supported in Windows XP or in Windows Server 2003.

In a fan-in configuration, the user can specify a connection URI and an HTTP endpoint, including the transport, computer name, port, and application name. IIS forwards all the requests with a specified application name to the application. The default is WS-Management, which can host PowerShell.

You can also specify an authentication mechanism and prohibit or allow redirection from HTTP and HTTPS endpoints.

Can I test remoting on a single computer not in a domain?

Yes. PowerShell remoting is available even when the local computer is not in a domain. You can use the remoting features to connect to sessions and to create sessions on the same computer. The features work the same as they do when you connect to a remote computer.

To run remote commands on a computer in a workgroup, change the following Windows settings on the computer.

Caution: These settings affect all users on the system and they can make the system more vulnerable to a malicious attack. Use caution when making these changes.

- Windows Vista, Windows 7, Windows 8:

Create the following registry entry, and then set its value to 1:

LocalAccountTokenFilterPolicy in

`HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System`

You can use the following PowerShell command to add this entry:

```
PowerShell

$parameters = @{

    Path='HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System'
    Name='LocalAccountTokenFilterPolicy'
    propertyType='DWord'
    Value=1
}

New-ItemProperty @parameters
```

- Windows Server 2003, Windows Server 2008, Windows Server 2012, Windows Server 2012 R2:

No changes are needed because the default setting of the "Network Access: Sharing and security model for local accounts" policy is "Classic". Verify the setting in case it has changed.

Can I run remote commands on a computer in another domain?

Yes. Typically, the commands run without error, although you might need to use the **Credential** parameter of the `Invoke-Command`, `New-PSSession`, or `Enter-PSSession` cmdlets to provide the credentials of a member of the Administrators group on the remote computer. This is sometimes required even when the current user is a member of the Administrators group on the local and remote computers.

However, if the remote computer is not in a domain that the local computer trusts, the remote computer might not be able to authenticate the user's credentials.

To enable authentication, use the following command to add the remote computer to the list of trusted hosts for the local computer in WinRM. Type the command at the PowerShell prompt.

```
PowerShell
```

```
Set-Item WSMAN:\localhost\Client\TrustedHosts -Value <Remote-computer-name>
```

For example, to add the Server01 computer to the list of trusted hosts on the local computer, type the following command at the PowerShell prompt:

```
PowerShell
```

```
Set-Item WSMAN:\localhost\Client\TrustedHosts -Value Server01
```

Does PowerShell support remoting over SSH?

Yes. For more information, see [PowerShell remoting over SSH](#).

See also

[about_Remote](#)

[about_Profiles](#)

[about_PSSessions](#)

[about_Remote_Jobs](#)

[about_Remote_Variables](#)

[Invoke-Command](#)

[New-PSSession](#)

Desired State Configuration (DSC) Overview

Article • 03/19/2025

DSC is a management platform that enables you to manage your IT and development infrastructure with configuration as code.

There are four versions of DSC available:

- **Microsoft DSC 3.0** is the new version of DSC. This version provides true cross-platform support. It is a standalone product that's not dependent on PowerShell, however, you can still use your existing PowerShell DSC resources.
- **PowerShell DSC 3.0 (preview)** is the version of DSC supported by the [Azure Machine Configuration](#) on Linux.
- **PowerShell DSC 2.0** is the version of DSC that shipped in PowerShell 7.

With the release of PowerShell 7.2, the **PSDesiredStateConfiguration** module is no longer included in the PowerShell package. Separating DSC into its own module allows us to invest and develop DSC independent of PowerShell and reduces the size of the PowerShell package. Users of DSC will enjoy the benefit of upgrading DSC without the need to upgrade PowerShell, accelerating the time to deployment of new DSC features. Users that want to continue using DSC v2 can download **PSDesiredStateConfiguration** 2.0.5 from the PowerShell Gallery.

- **PowerShell DSC 1.1** is the legacy version of DSC that originally shipped in Windows PowerShell 5.1.

For more information, see the [Desired State Configuration](#) overview article.

The PowerShell Gallery

Article • 04/13/2023

The [PowerShell Gallery](#) is the central repository for PowerShell content. In it, you can find PowerShell scripts, modules containing PowerShell cmdlets and Desired State Configuration (DSC) resources. Some of these packages are authored by Microsoft, and others are authored by the PowerShell community.

The **PowerShellGet** module contains cmdlets for discovering, installing, updating, and publishing PowerShell packages from the PowerShell Gallery. These packages can contain artifacts such as Modules, DSC Resources, Role Capabilities, and Scripts. Make sure you have the latest version of **PowerShellGet** installed.

The documentation for **PowerShellGet** and the PowerShell Gallery has been moved to a new location so that we can manage the version-specific information separate from the versions of PowerShell.

See the new documentation in [PowerShellGet and the PowerShell Gallery](#).



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Community Update

A list of resources and a summary of new articles and community contributions.

What's new in Docs?

WHAT'S NEW

[2025 Updates](#)

[2024 Updates](#)

[2023 Updates](#)

[2022 Updates](#)

[2021 Updates](#)

[2020 Updates](#)

[Contributor Hall of Fame](#)

Learning resources

TRAINING

[PowerShell 101](#)

[Deep Dives](#)

[PowerShell Learn modules](#)

VIDEO

[Microsoft Virtual Academy videos](#)

[Jason Helmick's - The Show ↗](#)

[The PSConfEU Channel ↗](#)

[The PowerShell.org Channel ↗](#)

Community resources

 OVERVIEW

[Community support](#)

 DOWNLOAD

[Digital art](#)

 REFERENCE

[PowerShell 7 usage stats ↗](#)

[Top contributors to PowerShell ↗](#)

Getting support from the community

Article • 03/30/2025

The PowerShell Community is a vibrant and active group of users. This article can help you get connected with other member of the community.

The PowerShell community can file issues, bugs, or feature requests in our [GitHub](#) repository. If you have questions, you may find help from other members of the community in one of these public forums:

- [User Groups](#)
- [PowerShell Tech Community](#)
- [DSC Community](#)
- [PowerShell.org](#)
- [Stack Overflow](#)
- [r/PowerShell subreddit](#)
- PowerShell Virtual User Group - join via:
 - [Slack](#)
 - [Discord](#)

For information about our support policy, see the [PowerShell Support Lifecycle](#).

Community Contributor Hall of Fame

06/02/2025

The PowerShell Community is a vibrant and collaborative group. We greatly appreciate all the help and support we get from the community. You can be a contributor too. To learn how, read our [Contributor's Guide](#).

These GitHub users are the All-Time Top Community Contributors.

Pull Requests merged

Pull Requests help us fix those issues and make the documentation better for everyone.

 Expand table

PRs Merged	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	Grand Total
vors	15	1										16
markekraus		11	5									16
kvprasoon	2	1	7	2	2	2						16
k-takai			5	1	7							13
purdo17			13									13
exchange12rocks		7	3				1					11
PlagueHO	10			1								11
bergmeister		1	3	3	1	1	1	1				11

GitHub issues opened

GitHub issues help us identify errors and gaps in our documentation.

[\[+\]](#) Expand table

Issues Opened	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	Grand Total
Community	3	54	95	211	562	557	363	225	270	221	86	2647
mklement0			19	60	56	61	28	8	20	24	2	278
ehmiiz							20	14				34
iSazonov		1	4	10	8	4	3		1			31
jszabo98			2	15	6	1		1	2			27
iRon7				2	2	2	10	8	1	25		
juvtib				15	7							22
doctordns	5	3	5	7	1							21
surfingoldelephant								6	14			20
peetrike			1		4	2	6	4	3			20
JustinGrote			1	3	6	1	1	2	2	2		18
vexx32	3	11						3				17

Issues Opened	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	Grand Total
kilasuit					3	2	1	4	1	3	2	16
KirkMunro				7	7	1						15
alexandair	9	4	2									15
clamb123						14						14
tabad								11	2			13
rkeithhill	1	2	2	2	3	1	2					13
trollyanov					11	1						12
jsilverm				8			4					12
CarloToso							11					11
Liturgist	1	1	1	1	1	1	2	4	2			11
vors	1	6	2	1								10
ArmaanMcleod							4	6				10
UberKluger					1	7	2					10
LaurentDardenne	3	2				5						10
matt9ucci	2	5			2		1					10

What's new in PowerShell Docs for 2025

06/02/2025

This article lists notable changes made to docs each month and celebrates the contributions from the community.

Help us make the documentation better for you. Read the [Contributor's Guide](#) to learn how to get started.

2025-May

Content updates

- Add new article - [Optimize performance using parallel execution](#)
- Updated release notes for the PowerShell 7.4.10 release
- Updated release notes for AIShell 1.0.0-preview.4
- Updated docs for PSResourceGet 1.1.1 release - no supporting MAR
- Retired PowerShell content for Windows Server 2021/2012r2/MDOP (5338 articles)

GitHub stats

- 27 PRs merged (10 from Community)
- 32 issues opened (31 from Community)
- 31 issues closed (30 Community issues closed)

Top Community Contributors

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

 Expand table

GitHub Id	PRs merged	Issues opened
changeworld	3	
cflanderscpc	1	
HeyItsGilbert	1	
dodekahedron	1	
MaratMussabekov	1	
geeksbsmrt	1	

GitHub Id	PRs merged	Issues opened
SP3269	1	
ajansveld	1	
robinmalik	1	
andreshungbz	1	
mklement0		2

2025-April

Content updates

- Updated release notes for the 7.5.1 and 7.6-preview.4 releases
- Refactored documentation of filter functions
- Documented bugs with CIM/CDXML bugs and PipelineVariable parameter

GitHub stats

- 24 PRs merged (6 from Community)
- 23 issues opened (22 from Community)
- 23 issues closed (23 Community issues closed)

Top Community Contributors

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[Expand table](#)

GitHub Id	PRs merged	Issues opened
surfingoldelephant	3	1
IISResetMe	1	
changeworld	1	
indented-automation	1	

2025-March

Content updates

- DSC v3 GA release - Complete reorg of the documentation as well as updating for the GA release
- PSScriptAnalyzer v1.24.0 release - updated docs for release
- Tons for quality improvements from surfingoldelephant
 - 38 PRs cleaning up 1505 files

GitHub stats

- 68 PRs merged (40 from Community)
- 30 issues opened (28 from Community)
- 30 issues closed (28 Community issues closed)

Top Community Contributors

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
surfingoldelephant	38	2
Allyyyyy	1	
jborean93	1	
JustinGrote		2

2025-February

Content updates

- Updated the Microsoft Update FAQ
- Tons of quality improvements from the community
 - surfingoldelephant - 13 PRs on 497 files
 - ArieHein - 10 PRs on 140 files

GitHub stats

- 52 PRs merged (29 from Community)
- 26 issues opened (25 from Community)
- 28 issues closed (26 Community issues closed)

Top Community Contributors

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[] Expand table

GitHub Id	PRs merged	Issues opened
surfingoldelephant	13	3
ArieHein	10	
changeworld	4	
deadlydog	2	
piedquance		4

2025-January

New content

- [about_Comments](#) - thanks to @surfingoldelephant
- Created reference content for PowerShell 7.6-preview
- [What's New in PowerShell 7.6](#)

Updates

- Updates for PowerShell 7.5.0 GA release
 - [What's New in PowerShell 7.5](#)
 - [Release history of modules and cmdlets](#)
- Special thanks to @ArieHein for his contributions (15 PRs on 234 files) to fix typos and adherence to style guidelines.

GitHub stats

- 74 PRs merged (30 from Community)
- 32 issues opened (32 from Community)
- 37 issues closed (37 Community issues closed)

Top Community Contributors

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[] Expand table

GitHub Id	PRs merged	Issues opened
ArieHein	15	1
changeworld	8	
surfingoldelephant	4	5
cnotin	1	
daniel-brandenburg	1	
o-l-a-v	2	

What's new in PowerShell Docs for 2024

Article • 03/30/2025

This article lists notable changes made to docs each month and celebrates the contributions from the community.

Help us make the documentation better for you. Read the [Contributor's Guide](#) to learn how to get started.

2024-December

- Added [about_Type_Conversion](#)
- Updated and improved documentation for several PSScriptAnalyzer rules
 - [AvoidDefaultValueSwitchParameter](#)
 - [AvoidGlobalFunctions](#)
 - [AvoidOverwritingBuiltInCmdlets](#)
 - [AvoidUsingCmdletAliases](#)
 - [AvoidUsingWriteHost](#)
 - [PlaceOpenBrace](#)
 - [PossibleIncorrectComparisonWithNull](#)
 - [PossibleIncorrectUsageOfAssignmentOperator](#)
 - [ProvideCommentHelp](#)
 - [UseApprovedVerbs](#)
 - [UseCompatibleCmdlets](#)
 - [UseCompatibleCommands](#)
 - [UseCompatibleTypes](#)
 - [UseShouldProcessForStateChangingFunctions](#)
 - [UseSupportsShouldProcess](#)

Top Community Contributors

GitHub stats

- 22 PRs merged (7 from Community)
- 30 issues opened (30 from Community)
- 24 issues closed (24 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[\[\] Expand table](#)

GitHub Id	PRs merged	Issues opened
sethvs	3	
ArieHein	1	
bharathalleni	1	
jhribal	1	
Saibamen	1	
skycommand	1	
surfingoldelephant		6
ArmaanMcleod		2
dpareit		2

2024-November

New and updated content

- 15 new article about AI Shell - [What is AI Shell?](#)
- Updated [What's New in PowerShell 7.5](#) for the RC1 release

Top Community Contributors

GitHub stats

- 21 PRs merged (9 from Community)
- 14 issues opened (14 from Community)
- 14 issues closed (14 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[\[\] Expand table](#)

GitHub Id	PRs merged	Issues opened
ArieHein	5	
abeu1	1	

GitHub Id	PRs merged	Issues opened
alexandair	1	
igoragoli	1	
jmillerca	1	
SamB	1	
uiolee	1	1

2024-October

New content

- [Improve the accessibility of DSC output in PowerShell](#)
- Added cmdlet reference for the [Microsoft.PowerShell.PlatyPS Module v1.0.0-preview1](#) release

Top Community Contributors

GitHub stats

- 30 PRs merged (11 from Community)
- 22 issues opened (22 from Community)
- 22 issues closed (22 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[Expand table](#)

GitHub Id	PRs merged	Issues opened
jhribal	4	
ArieHein	2	
Bergbok	1	1
colinwebster-hc	1	
HotCakeX	1	
pmsjt	1	

GitHub Id	PRs merged	Issues opened
CameronSWilliamson	1	
FARICJH59		2

2024-September

New content

- [Improve the accessibility of output in PowerShell](#)
- [What's new in PSResourceGet?](#)
- [about_PSReadLine_Release_Notes](#)

Top Community Contributors

GitHub stats

- 29 PRs merged (1 from Community)
- 19 issues opened (19 from Community)
- 20 issues closed (20 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
iRon7		5
jikuja		2

2024-August

- Updates for new releases of PowerShell 7.2.23, 7.4.5 and 7.5-preview.4
- Two new cmdlets in 7.5-preview.4
 - [ConvertFrom-CliXml](#)
 - [ConvertTo-CliXml](#)

Top Community Contributors

GitHub stats

- 25 PRs merged (4 from Community)
- 20 issues opened (18 from Community)
- 20 issues closed (17 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[+] [Expand table](#)

GitHub Id	PRs merged	Issues opened
RokeJulianLockhart	1	
mohitNarang	1	
aberus	1	

2024-July

- Updates for new releases of PowerShell 7.2.22 and 7.4.4
- Improved registry examples in:
 - [Working with registry entries](#)
 - [about_Registry_Provider](#)

Top Community Contributors

GitHub stats

- 38 PRs merged (11 from Community)
- 20 issues opened (20 from Community)
- 24 issues closed (24 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[+] [Expand table](#)

GitHub Id	PRs merged	Issues opened
guillermooo	4	
darthwalsh	2	

GitHub Id	PRs merged	Issues opened
shekeriev	1	
Blake-Madden	1	
paaspaas00	1	
ninmonkey	1	1
o-l-a-v	1	

2024-June

New content

- [TabExpansion2](#)

Top Community Contributors

GitHub stats

- 35 PRs merged (5 from Community)
- 16 issues opened (13 from Community)
- 16 issues closed (13 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
mklement0		2
sethvS	2	
InvalidAccountNameEntered	1	
OS3DrNick	1	
trackd	1	
landon-lengyel	1	
msklv	1	

2024-May

Retired PowerShell 7.3 content

- Moved 467 articles to *Previous Versions PowerShell content archive*

Updated content for PowerShell 7.5-preview.3 release

- [What's New in PowerShell 7.5](#)

New content for DSC v3 alpha release

- See [Desired State Configuration changelog](#)

Top Community Contributors

GitHub stats

- 31 PRs merged (11 from Community)
- 22 issues opened (21 from Community)
- 19 issues closed (18 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[] Expand table

GitHub Id	PRs merged	Issues opened
berlintay	1	
dcarpenter31	1	
i5513		2
MatejKafka	1	
real-guanyuming-he	1	
santisq	2	
stephanadler1	1	
stevenjudd	2	

2024-Apr

New content for PSResourceGet

- [Supported repository configurations](#)
 - Added Azure Container Registry information
- [Use ACR repositories with PSResourceGet](#)

Refreshed and reorganized Security content to be more discoverable

- [PowerShell Security](#)

New content for DSC v3 alpha.7 release

- See [Desired State Configuration changelog](#)

Top Community Contributors

GitHub stats

- 31 PRs merged (11 from Community)
- 22 issues opened (21 from Community)
- 19 issues closed (18 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[] [Expand table](#)

GitHub Id	PRs merged	Issues opened
Blake-Madden	3	
Hrxn	2	
aberus	1	
nickcox	1	
kilasuit	1	
glenn-slayden	1	
darthwalsh	1	
landon-l8	1	
joshooaj	1	
mklement0		5

GitHub Id	PRs merged	Issues opened
RokeJulianLockhart		2

2024-Mar

- DSC v3.0-alpha.5 release
 - See the updated [Changelog](#)
- PSScriptAnalyzer 1.22.0 release - 3 new rules
 - [AvoidMultipleTypeAttributes](#)
 - [AvoidSemicolonsAsLineTerminators](#)
 - [AvoidUsingBrokenHashAlgorithms](#)
 - See the [CHANGELOG](#) ↗ for a complete list of updates

Top Community Contributors

GitHub stats

- 30 PRs merged (1 from Community)
- 31 issues opened (28 from Community)
- 30 issues closed (26 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
Hrxn	3	
muzimuzhi	1	
bergmeister	1	

2024-Feb

- PowerShell 7.5.0-preview.2 release
 - See [What's New in PowerShell 7.5](#)
- Lots of updates for the DSC v3-alpha.5 release including 6 new articles
 - See [Desired State Configuration changelog](#)

- Worked with the Windows team to publish preview content for [Windows Server 2025](#)
 - This work included changes to the reference content and to the module source code to resolve errors reported by `Update-Help`

Top Community Contributors

GitHub stats

- 31 PRs merged (6 from Community)
- 26 issues opened (23 from Community)
- 27 issues closed (25 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[\[\] Expand table](#)

GitHub Id	PRs merged	Issues opened
Certezalito	1	
sethvs	1	
chrisdent-de	1	
jborean93	1	
brucificus	1	
XPlantefeve	1	
mklement0		4

2024-Jan

- PowerShell 7.5.0-preview.1 release
 - See [What's New in PowerShell 7.5](#)
- Lots of updates for the DSC v3-alpha.4 release including 6 new articles
 - See [Desired State Configuration changelog](#)

Top Community Contributors

GitHub stats

- 37 PRs merged (6 from Community)
- 36 issues opened (35 from Community)
- 36 issues closed (34 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
b-rad15	1	
flcdrg	1	
holtkampjs	1	
JamesDBartlett3	1	
MuzaffarNurillaew	1	
ybeltrikov	1	
ArmaanMcleod		3
mklement0		3
deraeler		2
J0F3		2
tabad		2
wwilliams69		2

What's new in PowerShell Docs for 2023

Article • 03/30/2025

This article lists notable changes made to docs each month and celebrates the contributions from the community.

Help us make the documentation better for you. Read the [Contributor's Guide](#) to learn how to get started.

2023-Dec

Lots of minor updates but no new content. The Docs team is taking a break for the holidays.

Top Community Contributors

GitHub stats

- 24 PRs merged (4 from Community)
- 21 issues opened (18 from Community)
- 18 issues closed (15 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[] [Expand table](#)

GitHub Id	PRs merged	Issues opened
Blake-Madden	2	
shyguyCreate	1	
ryanperrymba	1	
jp2images	1	
mklement0		2

2023-Nov

Updated content

- Updated release notes for PowerShell 7.4.0 GA
 - [What's New in PowerShell 7.4 - PowerShell](#)
 - [PowerShell Gallery - Microsoft.PowerShell.WhatsNew 0.5.4](#)
 - [PowerShell SDK 7.4 reference - .NET API browser](#)
- Retired Windows PowerShell content to archive site
 - Added [What is Windows PowerShell?](#)
- Major updates to: [about_Classes](#) and [about_Enum](#)

New content

- New Class articles added
 - [about Classes Constructors](#)
 - [about Classes Inheritance](#)
 - [about Classes Methods](#)
 - [about Classes Properties](#)

Top Community Contributors

GitHub stats

- 37 PRs merged (7 from Community)
- 32 issues opened (31 from Community)
- 33 issues closed (31 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
jmiller76	2	
VShrubowich	1	
TEBandCo	1	
skycommand	1	
darthwalsh	1	
diecknet	1	
tabad		7
mklement0		2

GitHub Id	PRs merged	Issues opened
radkedan		2

2023-Oct

New content

- [Securing a restricted PowerShell remoting session](#)

Updated content

- Updated release notes for PowerShell 7.4-rc.1
- Updated release notes for PSReadLine GA 2.3.4
- Updated release notes for PSResourceGet 1.0.0

Docs platform changes

- Released new feedback experience at the bottom of each page

Top Community Contributors

GitHub stats

- 55 PRs merged (11 from Community)
- 41 issues opened (37 from Community)
- 37 issues closed (33 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[Expand table](#)

GitHub Id	PRs merged	Issues opened
baardhermansen	1	
diecknet	1	
dipstar	1	
ehmiiz	1	1
G2-Games	1	
indented-automation	1	

GitHub Id	PRs merged	Issues opened
jhribal	1	
Joeselorm	1	
matziq	1	
RAJU2529	1	
ThomasNieto	1	2
iRon7		2
JustinWebDev		2
o-l-a-v		2
tabad		4

2023-Sep

New content

- [How to create a feedback provider](#)

Updated Content

- Updated [What's new in PowerShell 7.4](#) for PowerShell 7.4-preview.6
- Documented the changes to search scope in [How to use the documentation](#)
- Updated [What's new in Crescendo 1.1](#) for the GA release
- Updated the setup scripts for supported Linux distributions
- Updated DSC v3 content for the alpha.3 release

Quality improvement project contributions

- @ehmiiz contributed 5 PRs to update 35 files

New Learn platform features

- Deployed the new feedback experience at that bottom of each page

Top Community Contributors

GitHub stats

- 52 PRs merged (9 from Community)

- 24 issues opened (22 from Community)
- 24 issues closed (23 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[+] Expand table

GitHub Id	PRs merged	Issues opened
ehmiiz	5	5
bubbletroubles	1	
JamesDBartlett3	1	
not-not-kevin	1	
skycommand	1	
jsilverm		3
mklement0		2

2023-Aug

New content

66 New articles for DSC v3 (alpha)

- See [Microsoft Desired State Configuration v3 overview](#) to get started with the new documentation

Updated Content

- Crescendo 1.1-RC1 release updates
 - [What's new in Crescendo 1.1](#)
- New PSResourceGet beta24 content updates
 - See [Supported repository configurations](#)
- Lots updates for the PowerShell 7.4-preview.5 release
 - [What's New in PowerShell 7.4 \(preview\)](#)
 - New features for 10 cmdlets
- Updated support status and installation notes for Raspberry Pi
 - [Community support for PowerShell on Linux - PowerShell](#)

Top Community Contributors

GitHub stats

- 29 PRs merged (3 from Community)
- 16 issues opened (12 from Community)
- 21 issues closed (17 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[] Expand table

GitHub Id	PRs merged	Issues opened
ehmiiz	1	1
crisman	1	
deadlydog	1	
JamesDBartlett3	1	

2023-Jul

Updated content

- 7.4-preview.4 [release notes](#)
- Add publish information to [Supported repository configurations](#)
- Updated the release notes shipped with [Get-WhatsNew](#) ↗
- Fixed invalid ///-comments in SDK API reference
- Updated the man page that ships in PowerShell for Linux and macOS

Top Community Contributors

GitHub stats

- 31 PRs merged (10 from Community)
- 21 issues opened (17 from Community)
- 17 issues closed (13 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

GitHub Id	PRs merged	Issues opened
crisman	2	3
TSanzo-BLE	2	
MilekJakub	1	
Atman-Shastri	1	
BraveJhawk	1	
coolhome	1	
johndward01	1	
lor3k	1	

2023-Jun

New Azure Cloud Shell content

- [Using Cloud Shell in an Azure virtual network](#)
- [Deploy Azure Cloud Shell in a VNET with quickstart templates](#)

New PowerShellGet v3 content

- [Supported repository configurations](#)
- Cmdlet reference for [Microsoft.PowerShell.PSResourceGet](#)
- Cmdlet reference for the [PowerShellGet compatibility module](#)

Lot of updates for the PowerShell 7.4-preview.4 release

- [What's New in PowerShell 7.4 \(preview\)](#)
- [Using Experimental Features in PowerShell](#)
- New features for 12 cmdlets

Quality Project contributions

- @XXLMandalorian013 contributed 3 PRs to update 2 files in the MicrosoftDocs/windows-docs-powershell repository

Top Community Contributors

GitHub stats

- 59 PRs merged (15 from Community)
- 44 issues opened (30 from Community)
- 47 issues closed (31 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[] [Expand table](#)

GitHub Id	PRs merged	Issues opened
thegraffix	4	
crisman	4	3
RAJU2529	1	
dstreefkerk	1	
vontompers	1	
mubed	1	
Frederisk	1	
khaffner	1	
noamper	1	
aksarben		4

2023-May

New content

- New cmdlet in 7.4 - [Get-SecureRandom](#)
- [Using Windows Defender Application Control](#)

Updated content

- [How to use the PowerShell documentation](#)
 - Added descriptions for various navigation elements on the site
 - Added information about how to use the new [Download PDF](#) feature
- [Exploring the Windows PowerShell ISE](#)
 - Added screenshots for the ISE user interface elements
- Update documentation for [Add-Member](#)
 - Corrected some parameter values

- Improved the description and added examples for the **SecondValue** parameter

Quality Project contributions

- @robderickson contributed 1 PR to update 10 files in the MicrosoftDocs/windows-docs-powershell repository

Top Community Contributors

GitHub stats

- 38 PRs merged (7 from Community)
- 24 issues opened (18 from Community)
- 20 issues closed (17 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
rwp0	1	
pronichkin	1	
Ooggle	1	
mavaddat	1	
IanKemp	1	
mcdonaldjc	1	
Brizio	1	
dotnvo	1	
r0bfr	1	
mklement0		3
aksarben		2
crisman		2

2023-April

New content

- [Handling errors in Crescendo](#)
- [Transforming arguments in Crescendo](#)

Updated content

- Updated release notes for PowerShell 7.4-preview.3
- Migrated the PowerShell Gallery and PowerShellGet docs to new location to enable version selectors for [PowerShellGet](#)

PowerShell Summit 2023 - Hack-a-Doc event

- We hosted a Hack-a-Doc event on April 27th. Special thanks to the following 19 people. They contributed 62 PRs to update 204 files in the [MicrosoftDocs/windows-powershell-docs](#) repository.

[+] Expand table

GitHub Id	name	Count of PRs	Count of file
RobBiddle	Robert Biddle	28	48
pbossman	Phil Bossman	1	27
ThomasNieto	Thomas Nieto	1	24
kevinCefalu	Kevin Cefalu	1	24
robderickson	Rob Derickson	4	17
Snozzberries	Michael Soule	13	16
Spoonsk	Joseph Gast	1	12
thedavecarroll	Dave Carroll	1	11
raynbowbrite	Vanda Paladino	2	7
majst32	Melissa Januszko	1	6
XXLMandalorian013	Drew McClellan	1	4
ThePoShWolf	Anthony Howell	1	1
mdowst	Matthew Dowst	1	1
thepowerstring		1	1
KevinMarquette	Kevin Marquette	1	1

GitHub Id	name	Count of PRs	Count of file
53883		1	1
zockan	Michael Svegmar	1	1
lanwench	Paula Kingsley	1	1
stevenjudd	Steven Judd	1	1
Grand Total		62	204

Top Community Contributors

GitHub stats

- 23 PRs merged (2 from Community)
- 16 issues opened (14 from Community)
- 17 issues closed (15 Community issues closed)

The following people contributed to PowerShell docs this month by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
NLZ	1	
Jonathan-Quilter	1	

2023-March

New content

- Create a class-based DSC Resource for Machine Configuration
- Using PSReadLine key handlers

Updated content

- Release notes for [PowerShell 7.4-preview.2](#)

Quality project updates from the community

- One of our top contributors [@ehmiiz](#) blogged about contributing to Docs

- [How to Learn Git, Markdown and PowerShell by Contributing to the PowerShell-Docs Repository ↗](#)

Top Community Contributors

GitHub stats

- 60 PRs merged (13 from Community)
- 44 issues opened (31 from Community)
- 50 issues closed (36 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

 [Expand table](#)

GitHub Id	PRs merged	Issues opened
skycommand	2	
martincostello	1	
iRon7	1	4
chrullrich	1	
FlintyLemming	1	
ehmiiz	1	
vvavrychuk	1	
bb-froggy	1	
BenjamimCS	1	
kirillkrylov	1	
bergmeister	1	
lizy14	1	
CarloToso		5
MartinGC94		2
rgl		2

2023-February

New Content

- Preventing script injection attacks (Thanks [@PaulHigin](#)!)

Content updates

- Major update to [about_PowerShell_Config](#)
- Update to [about_Logging_Non-Windows](#) for macOS instructions
- Major update to [Class-based DSC Resources](#) and other related articles for DSC v2

Quality project updates from the community

- Added alias information to 4 cmdlet articles (Thanks [@ehmiiz](#)!)

Top Community Contributors

GitHub stats

- 35 PRs merged (8 from Community)
- 20 issues opened (14 from Community)
- 17 issues closed (10 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
joshua-russell	4	1
1NF053C	1	
doctordns	1	
Hrxn	1	
KyleMit	1	
VertigoRay	1	
ehmiiz	1	
ArmaanMcleod		2
mklement0		2

2023-January

New Content

- [What's new in PowerShell 7.4 \(preview\)](#)
- [about_Data_Files](#)

Content updates

- Updated docs for 7.4-preview.1 release
- Major update to [about_Language_Modes](#)
- Major update to [about_Logging_Non-Windows](#)

Quality project updates from the community

- Added alias information to 40 cmdlet articles (Thanks [@ehmiiz](#)!)
- Added alias information to 52 cmdlet articles (Thanks [@szabolevo](#)!)

Top Community Contributors

GitHub stats

- 71 PRs merged (21 from Community)
- 53 issues opened (33 from Community)
- 62 issues closed (40 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[Expand table](#)

GitHub Id	PRs merged	Issues opened
ehmiiz	5	5
szabolevo	5	7
anderjef	2	
turbedi	1	
desk7	1	
cobrabr	1	
ZYinMD	1	

GitHub Id	PRs merged	Issues opened
tompazourek	1	
kenyon	1	
cjvandyk	1	
JTBrinkmann	1	
mklement0		4
CarloToso		3
KyleMit		2
iRon7		2

What's new in PowerShell Docs for 2022

Article • 03/30/2025

This article lists notable changes made to docs each month and celebrates the contributions from the community.

Help us make the documentation better for you. Read the [Contributor's Guide](#) to learn how to get started.

2022-December

New Content

- [about_PSItem](#)
- [Configuring a light colored theme](#)
- [What's new in Crescendo 1.1](#)
- [Export-CrescendoCommand](#)
- PowerShell 7.4 (preview) cmdlet reference - a direct copy of the 7.3 content in preparation for the preview release of PowerShell 7.4

More Quality project updates

- Added alias information to 83 cmdlet articles (Thanks @ehmiiz!)
- Added alias information to 8 cmdlet articles (Thanks @szabolevo!)

GitHub stats

- 51 PRs merged (14 from Community)
- 50 issues opened (28 from Community)
- 46 issues closed (23 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[] Expand table

GitHub Id	PRs merged	Issues opened
ehmiiz	8	7
changeworld	3	
szabolevo	1	

GitHub Id	PRs merged	Issues opened
amkhrjee	1	
xtqqczze	1	3
ALiwoto	1	2
mklement0		3

2022-November

New Content

- [Contributing quality improvements](#)
 - See examples under [Quality project updates](#)
- [Product terminology and branding guidelines](#)
- [Labelling in GitHub](#)

Content updates

- Updated release notes for the PowerShell 7.3 GA release
- Updated [about_Telemetry](#)
- Improved the description of delay-binding in [about_Script_Blocks](#)
- Added a best practice recommendation to [about_Functions_Advanced_Parameters](#)

Quality project updates

- Added alias information to 129 cmdlet articles (Thanks @ehmiiz!)
- Added links to PRs in the PowerShell 7.3 release notes (Thanks @skycommand!)
- Converted hyperlinks to link references in 5 articles (Thanks @chadmando!)

GitHub stats

- 52 PRs merged (12 from Community)
- 41 issues opened (27 from Community)
- 42 issues closed (28 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[\] Expand table](#)

GitHub Id	PRs merged	Issues opened
ehmiiz	9	8
chadmando	1	
baardhermansen	1	
skycommand	1	
mklement0		3
peetrike		2

2022-October

New Content

- [Create a class-based DSC Resource](#)
- [Hacktoberfest and other hack-a-thon events](#)

Content updates

- [Hacktoberfest 2022](#) cleanup efforts
 - Thank you to @ehmiiz, @TSanzo-BLE, and @chadmando for their Hacktoberfest PRs! Their 11 PRs touched 114 articles.
- Published PowerShell SDK .NET API content for [PowerShell 7.2](#) and [7.3-preview](#)
 - The first updates since PowerShell 7.1 released in November 2020
 - Removed the unsupported versions 6.0 and 7.1
- Added a list of aliases not available on Linux and macOS to [PowerShell differences on non-Windows platforms](#)

GitHub stats

- 65 PRs merged (21 from Community)
- 42 issues opened (28 from Community)
- 34 issues closed (23 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[\] Expand table](#)

GitHub Id	PRs merged	Issues opened
ehmiiz	5	5

GitHub Id	PRs merged	Issues opened
TSanzo-BLE	4	
yecril71pl	2	
chadmando	2	
GigaScratch	1	1
rbleattler	1	
spjeff	1	
adamdriscoll	1	
manuelcarriernunes	1	
michelangelobottura	1	
dmpe	1	
KamilPacanek	1	
SetTrend		2

2022-September

No new content this month.

Content updates

- [PSScriptAnalyzer 1.21 update release](#)
- Release notes for [7.3-preview.8](#)

GitHub stats

- 31 PRs merged (8 from Community)
- 18 issues opened (9 from Community)
- 16 issues closed (8 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
DonaldDWebster	1	
emerconghaile	1	
floojah	1	
imere	1	
mcdonaldjc	1	
rayden84	1	
sirsql	1	
tig	1	
b-long		2

2022-August

New content

- New [landing page](#) for What's new content
- Shell experience docs
 - [Running commands in the shell](#)
- DSC 2.0 docs
 - [Conceptual content](#) - 15 new articles
 - [PSDscResources](#) module reference - 58 new articles

Content updates

- [PowerShell VS Code docs](#)
- Release notes for [7.3-preview.7](#)
- Cleaned up markdown tables in About topics for better accessibility and localization

Other Projects

- [Get-WhatsNew](#) cmdlet released - This cmdlet displays release notes for all versions of PowerShell so you can see what's new for a particular version.

GitHub stats

- 57 PRs merged (16 from Community)

- 24 issues opened (12 from Community)
- 26 issues closed (15 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[+] Expand table

GitHub Id	PRs merged	Issues opened
sethvsi	4	
BEEDELLROKEJULIANLOCKHART	1	
Chemerevsky	1	
ClaudioESSilva	1	
davidhaymond	1	
DavidMetcalfe	1	
dharmatech	1	
kozhemyak	1	
mcaawai	1	
NaridaL	1	
Nicicalu	1	
sdarwin	1	
seansaleh	1	

2022-July

New content

- [Optimizing your shell experience](#)
- [Using tab completion](#)
- [Using command predictors](#)
- [Getting dynamic help](#)
- [Using aliases](#)
- [Customizing your shell environment](#)

Content updates

- Updated PowerShell 7.3-preview.6 [release notes](#)
- Started reviewing and testing PowerShellGet v3 cmdlet reference (currently in beta) to ensure accuracy and release readiness.
- Refresh of our Community [Contributor Guide](#)

Top Community Contributors

GitHub stats

- 50 PRs merged (6 from Community)
- 22 issues opened (14 from Community)
- 29 issues closed (19 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[] [Expand table](#)

GitHub Id	PRs merged	Issues opened
Alvynskio	1	
bergmeister	1	
BusHero	1	
lewis-yeung	1	
sethvs	1	
tommymaynard	1	

2022-June

New content migrated from GitHub wiki

- [Limitations of PowerShell transcripts](#)
- [Avoid using Invoke-Expression](#)
- [Avoid assigning variables in expressions](#)
- [about_Case-Sensitivity](#)
- Updated [about_Arrays](#)

New SecretManagement content

- [Understanding the security features of SecretManagement and SecretStore](#)

- Using the SecretStore in automation
- Using Azure Key Vault in automation

Content updates

- Updated release notes for 7.3-preview.5 and PSReadLine 2.2.6

Top Community Contributors

GitHub stats

- 44 PRs merged (8 from Community)
- 23 issues opened (14 from Community)
- 23 issues closed (13 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[] Expand table

GitHub Id	PRs merged	Issues opened
mcdonaldjc	2	1
radrow	1	
yecril71pl	1	
muhahaaa	1	
windin7cc	1	
fabiod89	1	
NaridaL	1	

2022-May

New content

- Create a Crescendo configuration using the Crescendo cmdlets
- Overview of the SecretManagement and SecretStore modules
- Get started with the SecretStore module
- Understanding the SecretManagement module
- Managing a SecretStore vault

Content updates

- Renamed the `staging` branch to `main`
- Updated the Table of Contents for easier discovery
 - Moved Support Lifecycle to the top level
 - Moved Contributor Guide to the top level
- 7.3-preview.4 release notes
- Bulk formatting cleanup for many docs
 - PowerShell-Docs content - 272 files
 - Secrets management - 17 files
- Updated the PSScriptAnalyzer README and deleted docs that were migrated to Microsoft Learn
- Removed CentOS and Fedora from docs - no longer supported
- Retired 7.1 content - no longer supported
 - Collapse release notes into diff article
 - Delete or move content to archive repo

Top Community Contributors

GitHub stats

- 53 PRs merged (12 from Community)
- 38 issues opened (21 from Community)
- 39 issues closed (26 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[] Expand table

GitHub Id	PRs merged	Issues opened
tommymaynard	5	
naveensrinivasan	2	
rikurauhala	1	
joshua6point0	1	
rhorber	1	
Raton-Laveur	1	
StephenRoille	1	

GitHub Id	PRs merged	Issues opened
krlinus		2

2022-April

New content

- No new content this month

Content updates

- Rewrote the install instructions for [PowerShellGet](#)
- Created separate article for [Installing PowerShellGet on older Windows systems](#)

Other projects

- PowerShell + DevOps Summit April 25-28
 - Gave presentation about contributing to Docs
 - Lightning demo about argument completers
 - Interview for the [PowerShell Podcast](#) ↗

Top Community Contributors

GitHub stats

- 24 PRs merged (3 from Community)
- 22 issues opened (17 from Community)
- 21 issues closed (15 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[\]](#) Expand table

GitHub Id	PRs merged	Issues opened
Hrxn	1	
kevinholtkamp	1	
MikeyBronowski	1	
tommymaynard		4

2022-March

New Content

- New PowerShell docs
 - [about_Member-Access_Enumeration](#)
 - [about_Module_Manifests](#)
 - [How to create a command-line predictor](#)
- Utility modules updates
 - New docs for Crescendo release
 - [Install Crescendo](#)
 - [Choose a command-line tool](#)
 - [Decide which features to amplify](#)
 - [Create a Crescendo cmdlet](#)
 - [Generate and test a Crescendo module](#)
 - Moved PlatyPS article from PowerShell docs to the PlatyPS documentation
 - [Moved PlatyPS article](#)
 - Migrated more PSScriptAnalyzer documentation from the source code repository
 - [Using PSScriptAnalyzer](#)
 - [Rules and recommendations](#)
 - [Creating custom rules](#)

Content updates

- Bulk cleanup of related links in About_ topics
- Added issue and PR templates to all docs repos
- Updates for 7.3 preview content
 - New tab completions
 - Support for SSH options on remoting cmdlets
 - New experimental feature `PSAMSIMethodInvocationLogging`

Other projects

- Created a prototype cmdlet `Get-WhatsNew` based on the [draft RFC ↗](#)
 - Check out the RFC and provide feedback

New team member

- Welcome [Mikey Lombardi ↗](#) to the docs team

Top Community Contributors

GitHub stats

- 49 PRs merged (8 from Community)
- 26 issues opened (14 from Community)
- 33 issues closed (18 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
AspenForester	1	
codaamok	1	
DianaKuzmenko	1	
MikeyBronowski	1	
poshdude	1	
robcmo	1	
sertdfyguhi	1	
stampycode	1	

2022-February

New Content

- [about_Calling_Generic_Methods](#)

Content updates

- Catching up on issues
- Updates for 7.3 preview content

Top Community Contributors

GitHub stats

- 22 PRs merged (3 from Community)
- 24 issues opened (19 from Community)

- 18 issues closed (16 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
sethvs	2	
guilhermgonzaga	1	

2022-January

New Content

- No new content. We're down to one writer for PowerShell. I was out of the office for half of December for vacation then half of January for COVID.

Content updates

- Catching up on issues
- Updates for 7.3 preview content

Top Community Contributors

GitHub stats

- 51 PRs merged (10 from Community)
- 29 issues opened (26 from Community)
- 46 issues closed (39 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
sethvs	3	
UberKluger	1	
MiguelDomingues	1	

GitHub Id	PRs merged	Issues opened
reZach	1	
Hertz-Hu	1	
julian-hansen	1	
Hrxn	1	
peteraritchie	1	

What's new in PowerShell Docs for 2021

Article • 03/30/2025

This article lists notable changes made to docs each month and celebrates the contributions from the community.

Help us make the documentation better for you. Read the [Contributor's Guide](#) to learn how to get started.

2021-December

New Content

- Added PowerShell 7.3-preview.1 [preview content]
- New DSC 3.0 content
 - [PowerShell Desired State Configuration overview](#)
 - [Manage configuration using PowerShell DSC](#)
 - [DSC Configurations](#)
 - [DSC Resources](#)

Content updates

- Moved Desired State Configuration [content](#) to new docset and repository
 - DSC is now being developed outside of the PowerShell product.
 - The move allows for better versioning of documentation for DSC.

Top Community Contributors

GitHub stats

- 24 PRs merged (4 from Community)
- 30 issues opened (26 from Community)
- 12 issues closed (7 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[] [Expand table](#)

GitHub Id	PRs merged	Issues opened
dAu6jARL	1	

GitHub Id	PRs merged	Issues opened
shriharshmishra	1	
a-sync	1	
bogdangrigg	1	

2021-November

New Content

- [about_Built-in_Functions](#)

Content updates

- PowerShell 7.2 GA documentation updates
- Update GitHub Issue and PR templates - piloting the new YAML-based forms for issues
- Updated Crescendo reference for Preview 4 release

Top Community Contributors

GitHub stats

- 48 PRs merged (13 from Community)
- 31 issues opened (24 from Community)
- 34 issues closed (28 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
matt9ucci	4	
yecril71pl	3	
tholabrk	1	
lukejjh	1	
Oechiih	1	

GitHub Id	PRs merged	Issues opened
bergmeister	1	
Hrxn	1	
jebeckham	1	

2021-October

New Content

- PSAnalyzer documentation
 - [Overview](#)
 - [Rules documentation](#)

Content updates

- Lots of general editorial and freshness updates across 450 files
- PowerShell 7.2-rc.1 documentation updates

Top Community Contributors

GitHub stats

- 49 PRs merged (12 from Community)
- 33 issues opened (30 from Community)
- 33 issues closed (32 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
doctordns	4	
diecknet	1	
Kagre	1	
KexyBiscuit	1	
JohnRoos	1	

GitHub Id	PRs merged	Issues opened
Zhu-Panda	1	
philanderson888	1	
BlackFalcons	1	
milaandahiya	1	
mklement0		2

2021-September

New Content

- SDK documentation
 - [How to validate an argument using a script](#)
 - [ValidateScript Attribute Declaration](#)
- Learning content
 - [PowerShell security features](#)

Content updates

- Install documentation - Did a complete rewrite of the setup documentation. There is now a separate article for each supported OS.
 - [Install on Windows](#)
 - [Install on macOS](#)
 - [Install on Linux](#)
 - [Alpine](#)
 - [CentOS](#)
 - [Debian](#)
 - [Fedora](#)
 - [Raspberry Pi OS](#)
 - [Red Hat Enterprise Linux](#)
 - [Ubuntu](#)
 - [Alternate install methods](#)
 - [Community supported Linux](#)
- Supporting documentation
 - [Using PowerShell in Docker](#)
 - [Arm Processor support](#)
 - [Microsoft Update for PowerShell FAQ](#)
 - [PowerShell Support Lifecycle](#)

- Reformatted and updated the PSScriptAnalyzer rules documentation. Next month we plan to publish these docs to Microsoft Learn.
 - [Rules documentation on GitHub ↗](#)
- Lots of general editorial updates across 4500 files
- PowerShell 7.2-preview.10 documentation updates

Top Community Contributors

GitHub stats

- 68 PRs merged (6 from Community)
- 32 issues opened (29 from Community)
- 49 issues closed (41 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[] [Expand table](#)

GitHub Id	PRs merged	Issues opened
RaghuRocks3	1	
Zhu-Panda	1	
Jaykul	1	
juvtib	1	
przmv	1	
mklement0		2

2021-August

New content

- [about_ANSI_Terminals](#)
- [about_PSCustomObject](#)

Content updates

- PowerShell 7.2-preview.9 documentation updates

Top Community Contributors

GitHub stats

- 66 PRs merged (14 from Community)
- 42 issues opened (30 from Community)
- 53 issues closed (38 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[+] [Expand table](#)

GitHub Id	PRs merged	Issues opened
AvrumFeldman	1	
benmccallum	1	
bitdeft		2
BraINstinct0	1	
diddledani	1	
doctordns	1	
gravitational	1	
homotechsual	1	
imba-tjd	1	
juvtib	1	
kozhemyak	1	
omarys	1	
ryandasilva2992	1	
sethvs	1	
sneakernuts	1	

2021-July

New content

- [about_Functions_Argument_Completion](#)
- [about_Tab_Expansion](#)

Content updates

- [about_Functions_Advanced_Parameters](#) - major updates and links to new articles
- PowerShell 7.2-preview.8 documentation updates

Top Community Contributors

GitHub stats

- 51 PRs merged (9 from Community)
- 56 issues opened (50 from Community)
- 59 issues closed (52 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[] [Expand table](#)

GitHub Id	PRs merged	Issues opened
rkeithhill	2	
juvtib	2	
SetTrend	1	
Rob-S	1	
xtqqczze	1	
akashdabhi03	1	
kurtmckee	1	
clamb123		13
mklement0		2
jerryKahnPerl		2

2021-June

New content

- [about_Intrinsic_Members](#)
- [about_Booleans](#)

Content updates

- Converted [about_Remote_FAQ](#) to new YAML format and moved to conceptual TOC
- Moved **PSDesiredStateConfiguration** out of 7.2 docs and into PowerShell-Docs-Modules
 - DSC is being removed from PowerShell to become an optional module that is loaded from the PowerShell Gallery
 - Long-term plan is to move all DSC documentation out of PowerShell-Docs into a new repository for DSC content
- Totally rewrote the PowerShell release notes to summarize the current state, making it easier for users to find the information without having to read every release note.
 - [Differences between Windows PowerShell 5.1 and PowerShell \(core\) 7.x](#)
 - [PowerShell differences on non-Windows platforms](#)
- PowerShell 7.2-preview.7 documentation updates

Top Community Contributors

GitHub stats

- 43 PRs merged (1 from Community)
- 36 issues opened (32 from Community)
- 49 issues closed (41 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[+] [Expand table](#)

GitHub Id	PRs merged	Issues opened
rkeithhill	1	
frenchiveruti		2
paulaustin-automutatio		2
ringerc		2
trollyanov		2
UberKluger		2

2021-May

New Content

- Migrated two articles from the Windows Server content to the PowerShell docset
 - [PowerShell scripting performance considerations](#)
 - [PowerShell module authoring considerations](#)
- Added [PowerShell Language Specification 3.0](#)
 - The specification document is available from the Microsoft Download Center as a [Microsoft Word document ↗](#).
- Updated content for PowerShell 7.2-Preview6 release
- Moved Samples under the Learn node in the Table of Contents

Top Community Contributors

GitHub stats

- 53 PRs merged (6 from Community)
- 37 issues opened (35 from Community)
- 39 issues closed (36 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[] [Expand table](#)

GitHub Id	PRs merged	Issues opened
kvprasoon	2	
jErprog	1	
aamnah	1	
BetaLyte	1	
TheNCuber	1	
trollyanov		6
Tarjei-stavanger		3
aungminko93750		3
SetTrend		2
cdichter		2
reuvygroovy		2

2021-April

New Content

- Published new Learn content [Write your first PowerShell code](#)
- Updated docs for PowerShell 7.2-preview.5
- Updated metadata on ~3300 articles in the Windows module documentation
 - Preparing for Windows Server 2022 release and fixing updateable help
 - This is still a work in progress

Top Community Contributors

GitHub stats

- 45 PRs merged (5 from Community)
- 42 issues opened (33 from Community)
- 55 issues closed (32 Community issues closed)

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
ealmonete32	1	
hananyajacobson	1	
MarcChasse	1	
Melvin-Abraham	1	
robertkruk	1	
MikeMM70		2

2021-March

New content

- Working on simplifying and expanding the [Overview](#) content
 - Added [What is a PowerShell command?](#)
- Started a new tutorial series - [PowerShell Bits](#)
 - [Discover PowerShell](#)

- Help content for PowerShell [utility modules](#)
 - Microsoft.PowerShell.Crescendo
 - Microsoft.PowerShell.SecretManagement
 - Microsoft.PowerShell.SecretStore
 - PlatyPS
 - PSScriptAnalyzer
- Working with the Windows team to update help for [Windows management modules](#)
 - Added content for Windows Server 2019 and Windows Server 2022 (preview)
 - Continuing to work on improving updateable help for these modules
- PowerShell 7.2-preview documentation updates

Top Community Contributors

GitHub stats

- 42 PRs merged (5 from Community)
- 63 issues opened (42 from Community)
- 47 issues closed

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
bl-ue	1	
briinary	1	
Lx	1	
matt9ucci	1	
kfasick	1	
mklement0		10
juvtib		6
BoJackem23		2

2021-February

New content

- PowerShell 7.2-preview documentation updates

Top Community Contributors

GitHub stats

- 40 PRs merged (12 from Community)
- 40 issues opened (30 from Community)
- 35 issues closed

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
bbodenmiller	1	
briinary	1	
exchange12rocks	1	
lvenBach	1	
jamiepinheiro	1	
jdoubleu	1	
LogicalToolkit	1	
matt9ucci	1	
mihir-ajmera	1	
revolter	1	
secretGeek	1	
springcomp	1	
Ayanmullick		2

2021-January

New content

- PowerShell 7.2-preview documentation updates

Top Community Contributors

GitHub stats

- 44 PRs merged (14 from Community)
- 46 issues opened (38 from Community)
- 35 issues closed

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[\] Expand table](#)

GitHub Id	PRs merged	Issues opened
AndreyMatrosov	4	
revolter	2	
cconrad	1	
Hrxn	1	
kilasuit	1	
NN---	1	
snickler	1	
vinian	1	
zeekbrown	1	
briinary		2
mklement0		2
plastikfan		2

What's new in PowerShell Docs for 2020

Article • 03/30/2025

This article lists notable changes made to docs each month and celebrates the contributions from the community.

Help us make the documentation better for you. Read the [Contributor's Guide](#) to learn how to get started.

2020-December

- Updated contributor guide
 - documented the `&preserve_view=true` query parameter for hyperlinks
 - documented cross-reference syntax for hyperlinks
 - added information about localization
- GitHub stats
 - 44 PRs merged (11 from Community)
 - 50 issues opened (43 from Community)
 - 50 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[+] Expand table

GitHub Id	PRs merged	Issues opened
doctordns	3	2
chadbaldwin	1	
dawedawe	1	
dumpvn	1	
kvprasoon	1	
lbearl	1	
petershen0307	1	

GitHub Id	PRs merged	Issues opened
skycommand	1	
springcomp	1	
Cwilson-dataselfcom		5
bobbybatatina		2

2020-November

- PowerShell 7.1 GA Release
 - What's New in PowerShell 7.1
 - Converted 7.1 docs to release status
 - Added 7.2 (preview) docs
 - Retired v6 docs to archive repository
- Blog posts
 - [You've got Help!](#)
 - [Updating help for the PSReadLine module](#)
- Documentation maintenance
 - Updated 137 articles to remove MSDN and TechNet references
 - Updated 171 articles to indicate Windows-only compatibility
 - Updated 38 articles to address build warnings and suggestions
 - Added include to 24 DSC articles
 - Major rewrite of the PowerShell Jobs articles
 - [about_Jobs](#)
 - [about_Remote_Jobs](#)
 - [about_Thread_Jobs](#)
- GitHub stats
 - 50 PRs merged (8 from Community)
 - 55 issues opened (45 from Community)
 - 51 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

GitHub Id	PRs merged	Issues opened
AlanFlorance	1	
gilbertbw	1	
ianwalkeruk	1	
JeremyTBradshaw	1	
matt9ucci	1	
Rob-S	1	
ShaydeNofziger	1	
skycommand	1	
juvtib		8
iRon7		2
l-ip		2
stephenrgentry		2
Vixb1122		2

2020-October

- New articles
 - [about_Character_Encoding](#)
 - [about_Output_Streams](#)
 - [Using Visual Studio Code to debug compiled cmdlets](#) (thanks @fsackur)
 - [Add Credential support to PowerShell functions](#) (thanks @joshduffney)
- Documentation maintenance
 - Updates for 7.1-rc content
 - Updated all article descriptions to improve SEO
- GitHub stats
 - 61 PRs merged (7 from Community)
 - 49 issues opened (42 from Community)
 - 61 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[+] Expand table

GitHub Id	PRs merged	Issues opened
doctordns	1	
escape208	1	
nickdevore	1	
fsackur	1	
Duffney	1	
skycommand	1	
yecril71pl	1	
mklement0		3
Abdullah0820		2

2020-September

- Documentation maintenance
 - Updates for 7.1-preview content
- Community presentation
 - How to contribute to Docs for RTPUG - https://www.youtube.com/watch?v=0_DEB61YOMc
- GitHub stats
 - 41 PRs merged (9 from Community)
 - 52 issues opened (47 from Community)
 - 51 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[\] Expand table](#)

GitHub Id	PRs merged	Issues opened
doctordns	1	
fatherjack	1	
goforgold	1	
jonathanweinberg	1	
kvprasoon	1	
skycommand	1	
springcomp	1	
themichaelbender	1	
toddryan	1	
mklement0		13
setpeetrike		2

2020-August

- New PowerShell documentation
 - [About_Calculated_Properties](#)
 - [Writing Progress across multiple threads with ForEach-Object -Parallel](#)
- Documentation maintenance
 - Updates for 7.1-preview content
- GitHub stats
 - 69 PRs merged (26 from Community)
 - 68 issues opened (49 from Community)
 - 58 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[\[\] Expand table](#)

GitHub Id	PRs merged	Issues opened
sethvs	10	2
yecril71pl	10	
mklement0	1	7
springcomp	1	2
SquirrelAssassin		2
thorstenbutz		2
aetos382	1	
crumdev	1	
joshSi	1	
kmoad	1	

2020-July

- New PowerShell documentation
 - Resurrected old [ETS docs](#) - 7 articles added
 - Added article about [creating updateable help using PlatyPS](#)
- Documentation maintenance
 - Updates for 7.1-preview content
 - Updated page header - simplified menu choices and added a download button
 - Fixed several [Update-Help](#) issues
 - Help for PSDesiredStateConfiguration and ThreadJob modules now downloads
 - Published updateable help for PowerShell 7.1 preview
 - Updateable help for PowerShell 5.1 now includes About topics
- GitHub stats
 - 99 PRs merged (29 from Community)
 - 51 issues opened (44 from Community)
 - 71 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filing issues. Thank you!

[+] Expand table

GitHub Id	PRs merged	Issues opened
yecril71pl	10	3
sethvs	10	
springcomp	3	2
txtor	2	1
baardhermansen	1	
skycommand	1	
srjennings	1	
xtqqczze	1	
mklement0		3
Allexxann		2
sharpninja		2
XuHeng1021		2

2020-June

- New PowerShell documentation
 - Published new [PowerShell 101](#) content contributed by Mike F. Robbins
 - Added two recent blog posts from Rob Holt to the [Scripting and development](#) docs
 - [Choosing the right PowerShell NuGet package for your .NET project](#)
 - [Resolving PowerShell module assembly dependency conflicts](#)
- Documentation maintenance
 - Archived older content to <https://aka.ms/PSLegacyDocs>
 - PowerShell Web Access SDK content
 - PowerShell Workflows SDK content
 - Updates for 7.1-preview content
- GitHub stats

- o 83 PRs merged (15 from Community)
- o 68 issues opened (52 from Community)
- o 74 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filling issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
doctordns	3	
gabrielmcoll	2	
adrianhale	1	
aisbergde	1	
beatcracker	1	
bergmeister	1	
DarioArzaba	1	
gforceg	1	
jpomfret	1	
Karl-WE	1	
signalwarrant	1	
skycommand	1	
tkhadimullin	1	
johnkberry		2
juvtib		2
mklement0		2
Sagatboy33		4

2020-May

- New PowerShell documentation
 - Created a new [Deep dives](#) section containing content from community contributor Kevin Marquette
 - [Everything you want to know about arrays](#)
 - [Everything you want to know about hashtables](#)
 - [Everything you want to know about PSCustomObject](#)
 - [Everything you want to know about string substitution](#)
 - [Everything you want to know about if/then/else](#)
 - [Everything you want to know about switch](#)
 - [Everything you want to know about exceptions](#)
 - [Everything you want to know about \\$null](#)
 - [Everything you want to know about ShouldProcess](#)
 - [How to create a Standard Library binary module](#)
 - Published the [PowerShell 7.0 .NET API](#) reference
 - `Update-Help -Force` for PowerShell 5.1 now downloads updated content for the core PowerShell modules
- Documentation maintenance
 - Major reorganization of the Table of Contents
 - New content under **Learning PowerShell**
 - Windows PowerShell 5.1 content collected in one location
 - Archived older content to <https://aka.ms/PSLegacyDocs>
 - [PowerShell Web Access](#)
 - [PowerShell Workflows Guide](#)
 - Updates for 7.1-preview content
- GitHub stats
 - 81 PRs merged (21 from Community)
 - 61 issues opened (53 from Community)
 - 64 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filling issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
nschonni	10	

GitHub Id	PRs merged	Issues opened
xtqqczze	2	1
kilasuit	1	1
davidseibel	1	
doctordns	1	
jhoffmann271	1	
KevinMarquette	1	
klitztuch	1	
markojorge	1	
perjahn	1	
schuelermine	1	
jsilverm		7
mklement0		5
cam1170		2
JustinGrote		2
peetrike		2

2020-April

- New documents
 - PowerShell documentation
 - [Using Experimental Features](#)
 - [about_PSMODULEPATH](#)
 - Wiki articles
 - [The case for and against Invoke-Expression ↗](#)
 - [Variables can be assigned values as part of an expression \(with limitations\) ↗](#)
- Documentation maintenance
 - Now publishing updates to Microsoft Learn on a daily schedule
 - Monday-Friday 3pm Redmond Time (UTC-8)
 - Restructured the Community content
 - Many editorial cleanups
 - Updated Contributor Guide

- Clarified some formatting rules
- New information about table formatting
- GitHub stats
 - 74 PRs merged (8 from Community)
 - 79 issues opened (71 from Community)
 - 102 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filling issues. Thank you!

[\[+\] Expand table](#)

GitHub Id	PRs merged	Issues opened
ScSurber	1	1
alexandair	1	
bmkaiser	1	
hyoshioka0128	1	
jpomfret	1	
raydixon	1	
signalwarrant	1	
mklement0		8
reinierk		3
scabon		2
Abdullah0820		2
awooga		2
Damag3d		2
Fiasco123		2
Jasonthurston		2

2020-March

- New documents
 - The PowerShell Docs community pages
 - [Community resources](#) page
 - [What's new in PowerShell Docs](#) page (this page)
 - [PowerShell Infographic](#) ↗ added to the Digital Art page
 - [PowerShell-Doc contributor guide](#)
 - New PowerShell content
 - [Migrating from Windows PowerShell 5.1 to PowerShell 7](#)
 - [PowerShell 7 module compatibility list](#)
 - [Using PowerShell in Docker](#)
 - New Wiki content
 - [PowerShell prevents exceptions for non existent keys for types that implement IDictionary TKey, TValue](#) ↗
 - [PowerShell's treatment of namespaces is case insensitive but case preserving](#) ↗
- Documentation maintenance
 - Massive cleanup of broken links
 - Cleanup of old and duplicate issues
- GitHub stats
 - 100 PRs merged (14 from Community)
 - 68 issues opened (56 from Community)
 - 109 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filling issues. Thank you!

- k-takai - 7 PRs
- mklement0 - 5 issues
- juvtib - 4 issues
- iSazonov - 3 issue
- doctordns - 2 issues
- mdorantesm - 2 issues
- qt3m45su0najc7 - 2 issues

2020-February

- New documents
 - [about_Parameter_Sets](#)

- Release history of modules and cmdlets
- PowerShell 7 documentation updates
- Updates to address issues
- Ran PlatyPS to verify parameter info for all cmdlets and versions
- GitHub stats
 - 52 PRs merged (9 from Community)
 - 49 issues opened (42 from Community)
 - 55 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filling issues. Thank you!

- Maamue - 2 PRs
- doctordns - 2 PRs
- mklement0 - 4 issues
- he852100 - 3 issues
- VectorBCO - 2 issues
- metablaster - 2 issues

2020-January

- New documents
 - [about_Windows_PowerShell_Compatibility](#)
- PowerShell 7 documentation updates
- Updates to address issues
- GitHub stats
 - 58 PRs merged (7 from Community)
 - 57 issues opened (43 from Community)
 - 70 issues closed

Top Community Contributors

The following people have contributed to PowerShell docs by submitting pull requests or filling issues. Thank you!

- Makovec - 3 PRs
- mklement0 - 9 issues
- mvadu - 2 issues
- razos - 2 issues

- VLoub - 2 issues
- doctordns - 2 issues

PowerShell Digital Art

Article • 03/30/2025

The legends are true! The powerful shell that ensures safe passage to the cloud. But how?

Please enjoy the digital artwork linked below. Demonstrate to your peers that *you have been entrusted with the Scrolls of Monad!*

PowerShell Infographics

- [PowerShell Infographic ↗](#)

Comic

- [PowerShell Hero Comic \(High resolution\) ↗](#)
- [PowerShell Hero Comic \(Print resolution\) ↗](#)
- [PowerShell Hero Comic \(Web resolution\) ↗](#)

Wallpaper

- [PowerShell Hero Comic Wallpaper \(4k resolution\) ↗](#)
- [PowerShell Hero Pink Wallpaper \(4k resolution\) ↗](#)
- [PowerShell Hero White Wallpaper \(4k resolution\) ↗](#)

Poster

- [PowerShell Hero Poster ↗](#)

PowerShell Hero

- [PowerShell Hero Image ↗](#)

Microsoft PowerShell Logo and Digital Art Guidelines

As a general rule, third parties may not use Microsoft logos and artwork without permission. The following are the limited circumstances under which third parties may use the Microsoft PowerShell logo and artwork.

- For non-commercial purposes (documentation or on a website) that reference your connection with Microsoft PowerShell.

Any uses outside of these guidelines as determined by Microsoft is strictly prohibited.

Do not use the Microsoft PowerShell logo or artwork in products, product packaging, or other business services for which a formal license is required.

Microsoft reserves the right in its sole discretion to terminate or modify permission to display the logo or artwork, and may request that third parties modify or delete any use of the logo or artwork that, in Microsoft's sole judgment, does not comply with these guidelines or might otherwise impair Microsoft's rights in the logo.

Using Visual Studio Code for PowerShell Development

Article • 04/28/2023

[Visual Studio Code](#) (VS Code) is a cross-platform script editor by Microsoft. Together with the [PowerShell extension](#), it provides a rich and interactive script editing experience, making it easier to write reliable PowerShell scripts. Visual Studio Code with the PowerShell extension is the recommended editor for writing PowerShell scripts.

It supports the following PowerShell versions:

- PowerShell 7.2 and higher (Windows, macOS, and Linux)
- Windows PowerShell 5.1 (Windows-only) with .NET Framework 4.8

ⓘ Note

Visual Studio Code isn't the same as [Visual Studio](#).

Getting started

Before you begin, make sure PowerShell exists on your system. For modern workloads on Windows, macOS, and Linux, see the following links:

- [Installing PowerShell on Linux](#)
- [Installing PowerShell on macOS](#)
- [Installing PowerShell on Windows](#)

For traditional Windows PowerShell workloads, see [Installing Windows PowerShell](#).

ⓘ Important

The [Windows PowerShell ISE](#) is still available for Windows. However, it's no longer in active feature development. The ISE only works with PowerShell 5.1 and older. As a component of Windows, it continues to be officially supported for security and high-priority servicing fixes. we've no plans to remove the ISE from Windows.

Install VS Code and the PowerShell Extension

1. Install Visual Studio Code. For more information, see the overview [Setting up Visual Studio Code](#).

There are installation instructions for each platform:

- [Running Visual Studio Code on Windows](#)
- [Running Visual Studio Code on macOS](#)
- [Running Visual Studio Code on Linux](#)

2. Install the PowerShell Extension.

- a. Launch the VS Code app by typing `code` in a console or `code-insiders` if you installed Visual Studio Code Insiders.
- b. Launch **Quick Open** on Windows or Linux by pressing `Ctrl + P`. On macOS, press `Cmd + P`.
- c. In Quick Open, type `ext install powershell` and press **Enter**.
- d. The **Extensions** view opens on the Side Bar. Select the PowerShell extension from Microsoft.
- e. Click the **Install** button on the PowerShell extension from Microsoft.
- f. After the install, if you see the **Install** button turn into **Reload**, Click on **Reload**.
- g. After VS Code has reloaded, you're ready for editing.

For example, to create a new file, click **File > New**. To save it, click **File > Save** and then provide a filename, such as `HelloWorld.ps1`. To close the file, click the `X` next to the filename. To exit VS Code, **File > Exit**.

Installing the PowerShell Extension on Restricted Systems

Some systems are set up to require validation of all code signatures. You may receive the following error:

```
Language server startup failed.
```

This problem can occur when PowerShell's execution policy is set by Windows Group Policy. To manually approve PowerShell Editor Services and the PowerShell extension for VS Code, open a PowerShell prompt and run the following command:

```
PowerShell
```

```
Import-Module $HOME\.vscode\extensions\ms-
vscode.powershell*\modules\PowerShellEditorServices\PowerShellEditorServices
.psd1
```

You're prompted with **Do you want to run software from this untrusted publisher?** Type **A** to run the file. Then, open VS Code and verify that the PowerShell extension is functioning properly. If you still have problems getting started, let us know in a [GitHub issue](#).

Choosing a version of PowerShell to use with the extension

With PowerShell installing side-by-side with Windows PowerShell, it's now possible to use a specific version of PowerShell with the PowerShell extension. This feature looks at a few well-known paths on different operating systems to discover installations of PowerShell.

Use the following steps to choose the version:

1. Open the **Command Palette** on Windows or Linux with **ctrl + Shift + P**. On macOS, use **Cmd + Shift + P**.
2. Search for **Session**.
3. Click on **PowerShell: Show Session Menu**.
4. Choose the version of PowerShell you want to use from the list.

If you installed PowerShell to a non-typical location, it might not show up initially in the Session Menu. You can extend the session menu by [adding your own custom paths](#) as described below.

The PowerShell session menu can also be accessed from the **{}** icon in the bottom right corner of status bar. Hovering on or selecting this icon displays a shortcut to the session menu and a small pin icon. If you select the pin icon, the version number is added to the status bar. The version number is a shortcut to the session menu requiring fewer clicks.

Note

Pinning the version number replicates the behavior of the extension in versions of VS Code before 1.65. The 1.65 release of VS Code changed the APIs the PowerShell extension uses and standardized the status bar for language extensions.

Configuration settings for Visual Studio Code

First, if you're not familiar with how to change settings in VS Code, we recommend reading [Visual Studio Code's settings](#) documentation.

After reading the documentation, you can add configuration settings in `settings.json`.

JSON

```
{  
    "editor.renderWhitespace": "all",  
    "editor.renderControlCharacters": true,  
    "files.trimTrailingWhitespace": true,  
    "files.encoding": "utf8bom",  
    "files.autoGuessEncoding": true  
}
```

If you don't want these settings to affect all file types, VS Code also allows per-language configurations. Create a language-specific setting by putting settings in a `[<language-name>]` field. For example:

JSON

```
{  
    "[powershell)": {  
        "files.encoding": "utf8bom",  
        "files.autoGuessEncoding": true  
    }  
}
```

Tip

For more information about file encoding in VS Code, see [Understanding file encoding](#). Also, check out [How to replicate the ISE experience in VS Code](#) for other tips on how to configure VS Code for PowerShell editing.

Adding your own PowerShell paths to the session menu

You can add other PowerShell executable paths to the session menu through the [Visual Studio Code setting](#) `powershell.powerShellAdditionalExePaths`.

You can do this using the GUI:

1. From the **Command Palette** search for and select **Open User Settings**. Or use the keyboard shortcut on Windows or Linux `Ctrl + ,`. On macOS, use `Cmd + ,`.
2. In the **Settings** editor, search for **PowerShell Additional Exe Paths**.
3. Click **Add Item**.
4. For the **Key** (under **Item**), provide your choice of name for this additional PowerShell installation.

5. For the **Value** (under **Value**), provide the absolute path to the executable itself.

You can add as many additional paths as you like. The added items show up in the session menu with the given key as the name.

Alternatively you can add key-value pairs to the object

`powershell.powerShellAdditionalExePaths` in your `settings.json`:

JSON

```
{  
  "powershell.powerShellAdditionalExePaths": {  
    "Downloaded PowerShell":  
      "C:/Users/username/Downloads/PowerShell/pwsh.exe",  
    "Built PowerShell":  
      "C:/Users/username/src/PowerShell/src/powershell-win-  
      core/bin/Debug/net6.0/win7-x64/publish/pwsh.exe"  
  },  
}
```

ⓘ Note

Prior to version 2022.5.0 of the extension, this setting was a list of objects with the required keys `exePath` and `versionName`. A breaking change was introduced to support configuration via GUI. If you had previously configured this setting, please convert it the new format. The value given for `versionName` is now the **Key**, and the value given for `exePath` is now the **Value**. You can do this more easily by resetting the value and using the Settings interface.

To set the default PowerShell version, set the value

`powershell.powerShellDefaultVersion` to the text displayed in the session menu (the text used for the key):

JSON

```
{  
  "powershell.powerShellAdditionalExePaths": {  
    "Downloaded PowerShell":  
      "C:/Users/username/Downloads/PowerShell/pwsh.exe",  
  },  
  "powershell.powerShellDefaultVersion": "Downloaded PowerShell",  
}
```

After you've configured this setting, restart VS Code or to reload the current VS Code window from the **Command Palette**, type `Developer: Reload Window`.

If you open the session menu, you now see your additional PowerShell installations.

💡 Tip

If you build PowerShell from source, this is a great way to test out your local build of PowerShell.

Debugging with Visual Studio Code

No-workspace debugging

In VS Code version 1.9 (or higher), you can debug PowerShell scripts without opening the folder that contains the PowerShell script.

1. Open the PowerShell script file with **File > Open File...**
2. Set a breakpoint - select a line then press **F9**
3. Press **F5** to start debugging

You should see the Debug actions pane appear that allows you to break into the debugger, step, resume, and stop debugging.

Workspace debugging

Workspace debugging refers to debugging in the context of a folder that you've opened from the **File** menu using **Open Folder...**. The folder you open is typically your PowerShell project folder or the root of your Git repository. Workspace debugging allows you to define multiple debug configurations other than just debugging the currently open file.

Follow these steps to create a debug configuration file:

1. Open the **Debug** view on Windows or Linux by pressing **Ctrl + Shift + D**. On macOS, press **Cmd + Shift + D**.
2. Click the **create a launch.json file** link.
3. From the **Select Environment** prompt, choose **PowerShell**.
4. Choose the type of debugging you'd like to use:
 - **Launch Current File** - Launch and debug the file in the currently active editor window

- **Launch Script** - Launch and debug the specified file or command
- **Interactive Session** - Debug commands executed from the Integrated Console
- **Attach** - Attach the debugger to a running PowerShell Host Process

VS Code creates a directory and a file `.vscode\launch.json` in the root of your workspace folder to store the debug configuration. If your files are in a Git repository, you typically want to commit the `launch.json` file. The contents of the `launch.json` file are:

```
JSON

{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "PowerShell",
      "request": "launch",
      "name": "PowerShell Launch (current file)",
      "script": "${file}",
      "args": [],
      "cwd": "${file}"
    },
    {
      "type": "PowerShell",
      "request": "attach",
      "name": "PowerShell Attach to Host Process",
      "processId": "${command.PickPSPHostProcess}",
      "runspaceId": 1
    },
    {
      "type": "PowerShell",
      "request": "launch",
      "name": "PowerShell Interactive Session",
      "cwd": "${workspaceRoot}"
    }
  ]
}
```

This file represents the common debug scenarios. When you open this file in the editor, you see an **Add Configuration...** button. You can click this button to add more PowerShell debug configurations. One useful configuration to add is **PowerShell: Launch Script**. With this configuration, you can specify a file containing optional arguments that are used whenever you press `F5` no matter which file is active in the editor.

After the debug configuration is established, you can select the configuration you want to use during a debug session. Select a configuration from the debug configuration

drop-down in the **Debug** view's toolbar.

Troubleshooting the PowerShell extension

If you experience any issues using VS Code for PowerShell script development, see the [troubleshooting guide ↗](#) on GitHub.

Useful resources

There are a few videos and blog posts that may be helpful to get you started using the PowerShell extension for VS Code:

Videos

- [Using Visual Studio Code as Your Default PowerShell Editor ↗](#)
- [Visual Studio Code: deep dive into debugging your PowerShell scripts ↗](#)

Blog posts

- [PowerShell Extension](#)
- [Write and debug PowerShell scripts in Visual Studio Code ↗](#)
- [Debugging Visual Studio Code Guidance ↗](#)
- [Debugging PowerShell in Visual Studio Code ↗](#)
- [Get started with PowerShell development in Visual Studio Code ↗](#)
- [Visual Studio Code editing features for PowerShell development - Part 1 ↗](#)
- [Visual Studio Code editing features for PowerShell development - Part 2 ↗](#)
- [Debugging PowerShell script in Visual Studio Code - Part 1 ↗](#)
- [Debugging PowerShell script in Visual Studio Code - Part 2 ↗](#)

PowerShell extension project source code

The PowerShell extension's source code can be found on [GitHub ↗](#).

If you're interested in contributing, Pull Requests are greatly appreciated. Follow along with the [developer documentation ↗](#) on GitHub to get started.



Collaborate with us on
GitHub



PowerShell feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to replicate the ISE experience in Visual Studio Code

Article • 07/10/2023

While the PowerShell extension for VS Code doesn't seek complete feature parity with the PowerShell ISE, there are features in place to make the VS Code experience more natural for users of the ISE.

This document tries to list settings you can configure in VS Code to make the user experience a bit more familiar compared to the ISE.

ISE Mode

Note

This feature is available in the PowerShell Preview extension since version 2019.12.0 and in the PowerShell extension since version 2020.3.0.

The easiest way to replicate the ISE experience in Visual Studio Code is by turning on "ISE Mode". To do this, open the command palette (`F1` OR `Ctrl + Shift + P` OR `Cmd + Shift + P` on macOS) and type in "ISE Mode". Select "PowerShell: Enable ISE Mode" from the list.

This command automatically applies the settings described below. The result looks like this:

(PREVIEW) POWERSHELL COMMAND EXPLORER:... Untitled-1

```
Add-AzADGroupMember
Add-AzContainerServiceAgentPoolProfile
Add-AzEnvironment
Add-AzImageDataDisk
Add-AzKeyVaultCertificate
Add-AzKeyVaultCertificateContact
Add-AzKeyVaultKey
Add-AzKeyVaultManagedStorageAccount
Add-AzKeyVaultNetworkRule
Add-AzRmStorageContainerLegalHold
Add-AzStorageAccountManagementPolicyAct...
Add-AzStorageAccountNetworkRule
Add-AzVhd
Add-AzVMAdditionalUnattendContent
Add-AzVMDataDisk
Add-AzVMNetworkInterface
Add-AzVMSecret
Add-AzVmssAdditionalUnattendContent
Add-AzVmssDataDisk
Add-AzVmssDiagnosticsExtension
Add-AzVmssExtension
Add-AzVmssPublickey
Add-AzVmssNetworkInterfaceConfiguration
Add-AzVmssSecret
Add-AzVmssSshPublicKey
Add-AzVmssVMDataDisk
Add-AzVmssWinRMListener
Add-Content
Add-History
Add-Member
Add-PoshGitToProfile
Add-RoleMember
Add-Secret
Add-SqlAvailabilityDatabase
Add-SqlAvailabilityGroupListenerStaticIp
Add-SqlAzureAuthenticationContext
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

1: PowerShell Integrated Cor

```
===== PowerShell Integrated Console =====
tyler ~
>>> [ ]
```

ISE Mode configuration settings

ISE Mode makes the following changes to VS Code settings.

- Key bindings

Function	ISE Binding	VS Code Binding
Interrupt and break debugger	<code>Ctrl + B</code>	<code>F6</code>
Execute current line/highlighted text	<code>F8</code>	<code>F8</code>
List available snippets	<code>Ctrl + J</code>	<code>Ctrl + Alt + J</code>

! Note

You can **configure** your own key bindings in VS Code as well.

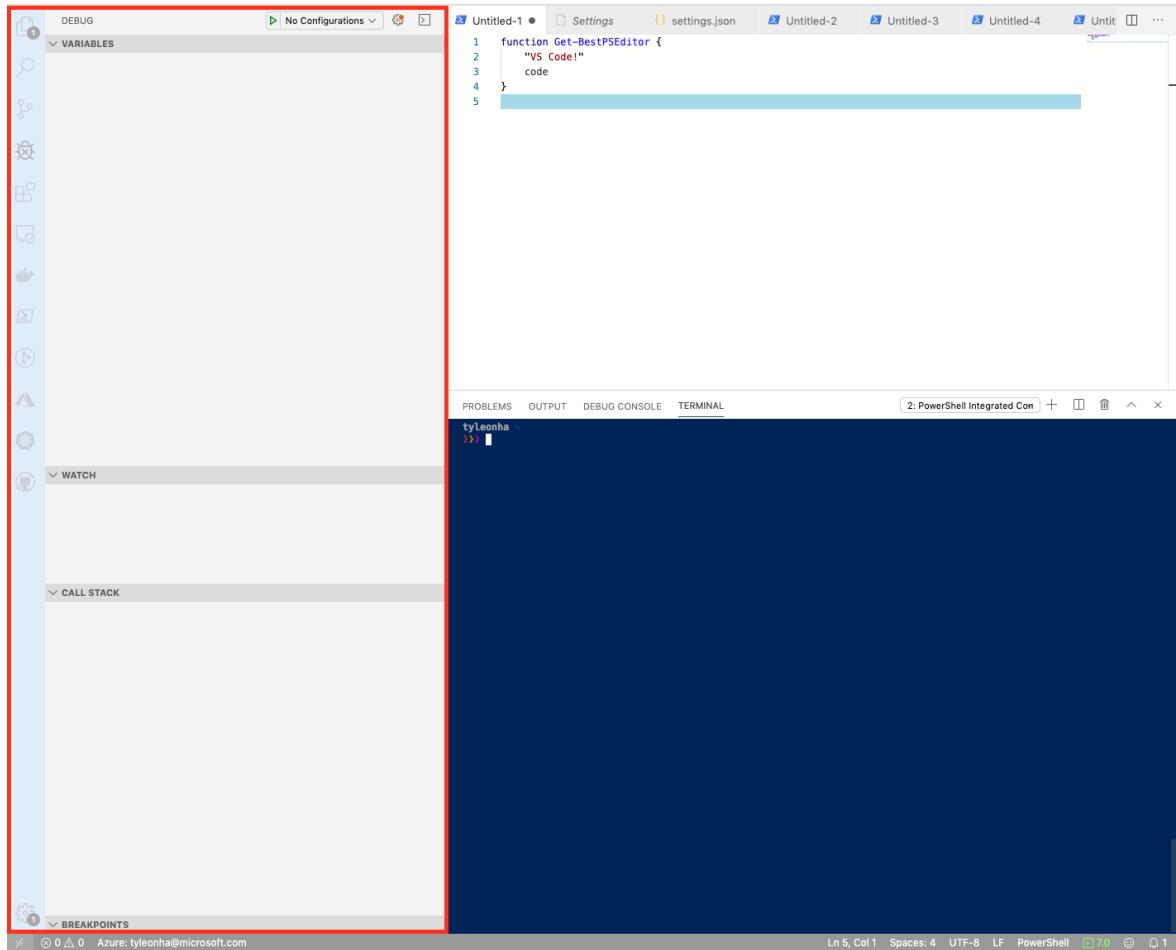
- Simplified ISE-like UI

If you're looking to simplify the Visual Studio Code UI to look more closely to the UI of the ISE, apply these two settings:

JSON

```
"workbench.activityBar.visible": false,  
"debug.openDebug": "neverOpen",
```

These settings hide the "Activity Bar" and the "Debug Side Bar" sections shown inside the red box below:



The end result looks like this:

A screenshot of the Microsoft Visual Studio Code interface. The top menu bar includes 'File', 'Edit', 'Selection', 'View', 'Go', 'Debug', 'Terminal', 'Window', and 'Help'. The title bar shows 'Code - Insiders' and the file 'Install-VSCode.ps1'. The main editor area displays a PowerShell script with code for installing VS Code. Below the editor are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The terminal tab is active, showing a dark blue background with white text. At the bottom of the screen, there are status indicators: 'tyleonha ~', '2: PowerShell Integrated Con', and various system status icons.

- Tab completion

To enable more ISE-like tab completion, add this setting:

JSON

```
"editor.tabCompletion": "on",
```

- No focus on console when executing

To keep the focus in the editor when you execute with **F8**:

JSON

```
"powershell.integratedConsole.focusConsoleOnExecute": false
```

The default is `true` for accessibility purposes.

- Don't start integrated console on startup

To stop the integrated console on startup, set:

JSON

```
"powershell.integratedConsole.showOnStartup": false
```

ⓘ Note

The background PowerShell process still starts to provide IntelliSense, script analysis, symbol navigation, etc., but the console won't be shown.

- Assume files are PowerShell by default

To make new/untitled files, register as PowerShell by default:

JSON

```
"files.defaultLanguage": "powershell",
```

- Color scheme

There are a number of ISE themes available for VS Code to make the editor look much more like the ISE.

In the [Command Palette](#) type `theme` to get [Preferences: Color Theme](#) and press `Enter`. In the drop-down list, select `PowerShell ISE`.

You can set this theme in the settings with:

JSON

```
"workbench.colorTheme": "PowerShell ISE",
```

- PowerShell Command Explorer

Thanks to the work of [@corbob](#), the PowerShell extension has the beginnings of its own command explorer.

In the [Command Palette](#), enter `PowerShell Command Explorer` and press `Enter`.

- Open in the ISE

If you want to open a file in the Windows PowerShell ISE anyway, open the [Command Palette](#), search for "open in ise", then select [PowerShell: Open Current](#)

Other resources

- 4sysops has [a great article ↗](#) on configuring VS Code to be more like the ISE.
- Mike F Robbins has [a great post ↗](#) on setting up VS Code.

VS Code Tips

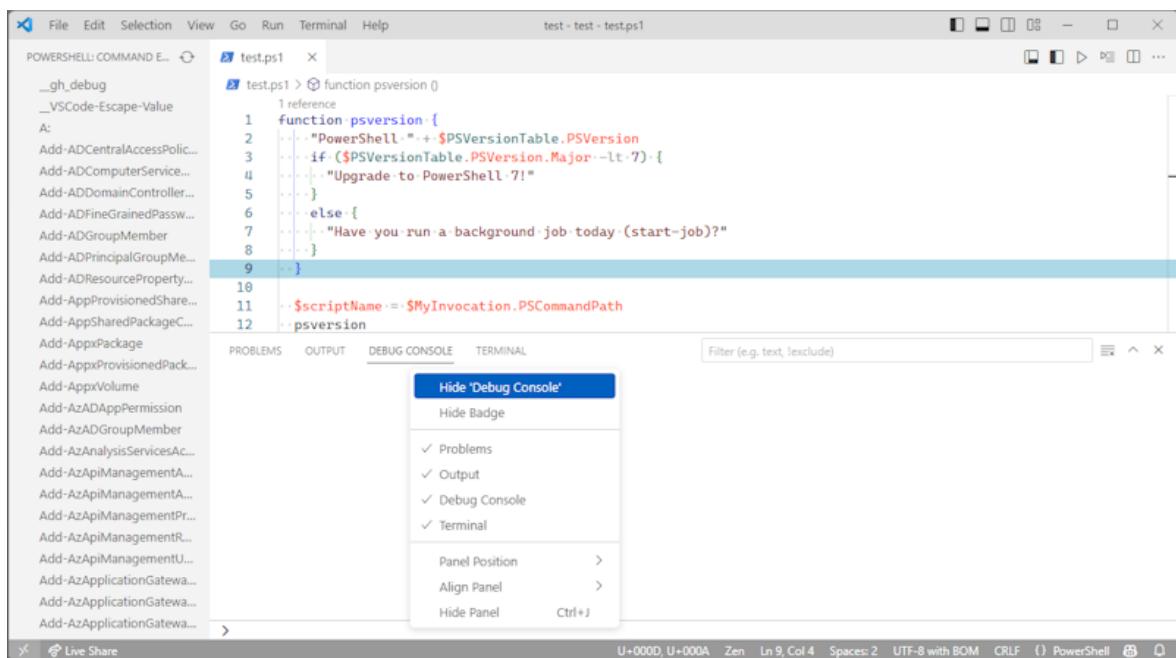
- Command Palette

The Command Palette is handy way of executing commands in VS Code. Open the command palette using **F1** OR **Ctrl + Shift + P** OR **Cmd + Shift + P** on macOS.

For more information, see [the VS Code documentation ↗](#).

- Hide the Debug Console panel

The PowerShell extension uses the built-in debugging interface of VS Code to allow for debugging of PowerShell scripts and modules. However, the extension does not use the Debug Console panel. To hide the Debug Console, right-click on **Debug Console** and select **Hide 'Debug Console'**.



For more information about debugging PowerShell with Visual Studio Code, see [Using VS Code](#).

More settings

If you know of more ways to make VS Code feel more familiar for ISE users, contribute to this doc. If there's a compatibility configuration you're looking for, but you can't find any way to enable it, [open an issue ↗](#) and ask away!

We're always happy to accept PRs and contributions as well!

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Using Visual Studio Code for remote editing and debugging

Article • 04/16/2025

For those of you that are familiar with the ISE, you may recall that you could run `psedit file.ps1` from the integrated console to open files - local or remote - right in the ISE.

This feature is also available in the PowerShell extension for VS Code. This guide shows you how to do it.

Prerequisites

This guide assumes that you have:

- A remote resource (ex: a VM, a container) that you have access to
- PowerShell running on it and the host machine
- VS Code and the PowerShell extension for VS Code

This feature works on PowerShell and Windows PowerShell.

This feature also works when connecting to a remote machine via WinRM, PowerShell Direct, or SSH. If you want to use SSH, but are using Windows, check out the [Win32 version of SSH](#)!

Important

The `Open-EditorFile` and `psedit` commands only work in the **PowerShell Integrated Console** created by the PowerShell extension for VS Code.

Usage examples

These examples show remote editing and debugging from a MacBook Pro to an Ubuntu VM running in Azure. The process is identical on Windows.

Local file editing with Open-EditorFile

With the PowerShell extension for VS Code started and the PowerShell Integrated Console opened, we can type `open-EditorFile foo.ps1` or `psedit foo.ps1` to open the local `foo.ps1` file right in the editor.

The screenshot shows a dark-themed instance of Visual Studio Code. In the top left, there are icons for file operations like Open, Save, and Close. The title bar shows the file name 'foo.ps1'. The main editor area contains the following PowerShell script:

```
1 Write-Host "psedit foo.ps1"
2
3 Write-Host "then you can edit this file"
4
5 Write-Host "and debug it too"
```

Below the editor, the bottom navigation bar includes tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is selected, showing the output of the command:

```
/Users/tylerleonhardt/Desktop> psedit ./foo.ps1
/Users/tylerleonhardt/Desktop> [ ]
```

The status bar at the bottom displays the file path, line count (Ln 5), column count (Col 30), space usage (Spaces: 4), encoding (UTF-8), line separator (LF), the current language mode (PowerShell), the commit hash (6.0), and a smiley face icon.

ⓘ Note

The file `foo.ps1` must already exist.

From there, we can:

- Add breakpoints to the gutter

The screenshot shows a dark-themed instance of Visual Studio Code. In the top-left corner, there are three window control buttons (red, yellow, green). The title bar displays "foo.ps1". On the left side, there's a vertical toolbar with icons for file operations, search, and other tools. The main editor area contains a PowerShell script named "foo.ps1" with the following content:

```
1 Write-Host "psedit foo.ps1"
2
3 Write-Host "then you can edit this file"
4
5 Write-Host "and debug it too"
```

Below the editor is a tab bar with "PROBLEMS", "OUTPUT", "DEBUG CONSOLE", and "TERMINAL". The "TERMINAL" tab is selected, showing the command "/Users/tylerleonhardt/Desktop> psedit ./foo.ps1" followed by a blank line. At the bottom of the screen, there's a purple status bar displaying "Ln 5, Col 30 Spaces: 4 UTF-8 LF PowerShell" and some other icons.

- Hit F5 to debug the PowerShell script.

The screenshot shows the PowerShell Integrated Terminal in Visual Studio Code. A breakpoint has been hit on line 3 of the script 'foo.ps1'. The terminal output shows the command 'psedit foo.ps1' being run, followed by the hit breakpoint message. The code editor on the left displays the script content:

```
1 Write-Host "psedit foo.ps1"
2
3 Write-Host "then you can edit this file"
4
5 Write-Host "and debug it too"
```

The terminal tab bar indicates the session is 1: PowerShell Int. The status bar at the bottom shows the current line (Ln 3, Col 41), spaces (Spaces: 4), encoding (UTF-8 LF), and PowerShell version (6.0).

While debugging, you can interact with the debug console, check out the variables in the scope on the left, and all the other standard debugging tools.

Remote file editing with Open-EditorFile

Now let's get into remote file editing and debugging. The steps are nearly the same, there's just one thing we need to do first - enter our PowerShell session to the remote server.

There's a cmdlet for to do so. It's called `Enter-PSSession`.

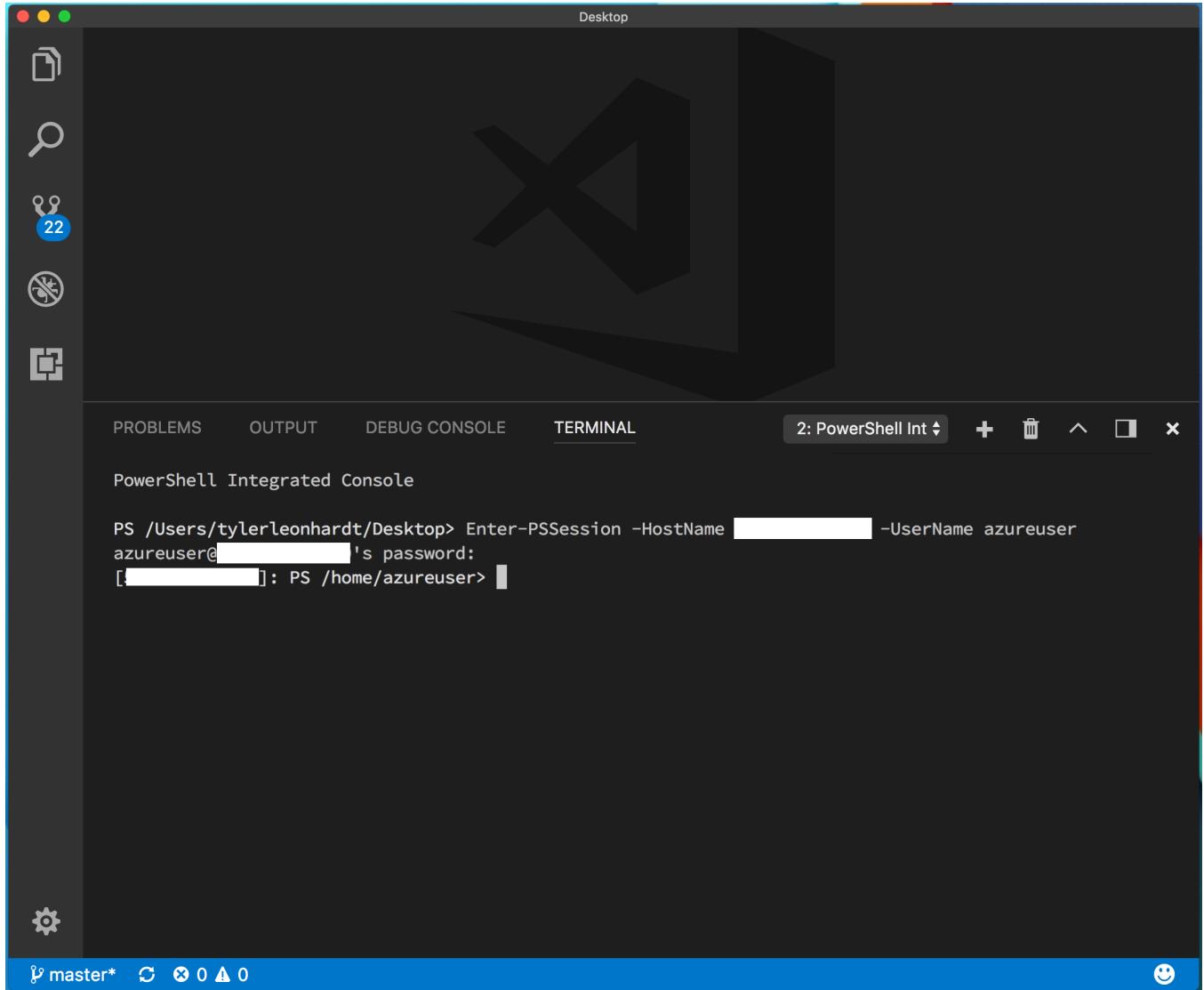
In short:

- `Enter-PSSession -ComputerName foo` starts a session via WinRM
- `Enter-PSSession -ContainerId foo` and `Enter-PSSession -VmId foo` start a session via PowerShell Direct
- `Enter-PSSession -HostName foo` starts a session via SSH

For more information, see the documentation for `Enter-PSSession`.

Since we're remoting to an Ubuntu VM in Azure, we're using SSH.

First, in the Integrated Console, run `Enter-PSSession`. You're connected to the remote session when [`<hostname>`] shows up to the left of your prompt.



The screenshot shows the Visual Studio Code desktop application window. The title bar says "Desktop". The left sidebar has icons for File, Search, Issues (22), and Snippets. The main area is a dark-themed terminal window titled "PowerShell Integrated Console". It displays the command `PS /Users/tylerleonhardt/Desktop> Enter-PSSession -HostName [REDACTED] -UserName azureuser`, followed by a password prompt: `azureuser@[REDACTED]'s password:`. Below the terminal, the status bar shows "Y master*" and "0 0 ▲ 0".

```
PowerShell Integrated Console

PS /Users/tylerleonhardt/Desktop> Enter-PSSession -HostName [REDACTED] -UserName azureuser
azureuser@[REDACTED]'s password:
[REDACTED]: PS /home/azureuser> 
```

Now, we can do the same steps as if we're editing a local script.

1. Run `Open-EditorFile test.ps1` or `psedit test.ps1` to open the remote `test.ps1` file

The screenshot shows the Visual Studio Code interface with the following details:

- Title Bar:** [Extension Development Host] - test.ps1 — Polaris
- Sidebar:** Includes icons for File, Find, Search, Diff, and Settings.
- Code Editor:** A file named "test.ps1" containing the following PowerShell code:

```
1 Write-Host "this file is on an Ubuntu VM in Azure"
2
3 Write-Host "It's running PowerShell Core"
4
5 Write-Host "Hello World!"
```
- Terminal:** An integrated PowerShell console titled "PowerShell Integrated Console". It shows the following session:

```
PS /Users/tylerleonhardt/Desktop/Polaris> Enter-PSSession -HostName [REDACTED] -UserName azureuser

azureuser@[REDACTED]'s password:
[REDACTED]: PS /home/azureuser> ls
test.ps1
[REDACTED]: PS /home/azureuser> PSEdit ./test.ps1
[REDACTED]: PS /home/azureuser> [REDACTED]
```
- Status Bar:** Shows the current branch as "master*", file count as "0 ▲ 0", line 5, column 25, spaces: 4, encoding as UTF-8, line feed as LF, and the language as PowerShell.

2. Edit the file/set breakpoints

The screenshot shows the Visual Studio Code interface with the following details:

- Title Bar:** [Extension Development Host] - test.ps1 — Polaris
- Code Editor:** A file named "test.ps1" is open, containing the following PowerShell script:

```
1 Write-Host "this file is on an Ubuntu VM in Azure"
2
3 Write-Host "It's running PowerShell Core"
4
5 Write-Host "Hello World!"
```
- Sidebar:** Includes icons for File, Search, Find in Current File, and a Taskbar with 22 items.
- Bottom Navigation:** PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (selected).
- Terminal:** PowerShell Integrated Console showing session activity:

```
PS /Users/tylerleonhardt/Desktop/Polaris> Enter-PSSession -HostName [REDACTED] -UserName azureuser
azureuser@[REDACTED]'s password:
[REDACTED]: PS /home/azureuser> ls
test.ps1
[REDACTED]: PS /home/azureuser> PSEdit ./test.ps1
[REDACTED]: PS /home/azureuser> [REDACTED]
```
- Bottom Status Bar:** master*, 0 errors, 0 warnings, Line 5, Column 25, Spaces: 4, UTF-8, LF, PowerShell, 6.0, [REDACTED], Smiley icon.

3. Start debugging (F5) the remote file

The screenshot shows the Polaris PowerShell IDE interface. On the left, there's a sidebar with icons for file operations, search, and debugging. The main area has tabs for 'DEBUG' and 'test.ps1'. The 'test.ps1' tab contains the following PowerShell script:

```
1 Write-Host "this file is on an Ubuntu VM in Azure"
2
3 Write-Host "It's running PowerShell Core"
4
5 Write-Host "Hello World!"
```

The line 'Write-Host "It's running PowerShell Core"' is highlighted in yellow, indicating it is the current line of execution. Below the code editor is a terminal window titled 'PowerShell Integrated Console' showing the command 'Enter-PSSession -HostName' followed by a redacted host name. The session is connected to an Ubuntu VM in Azure, as indicated by the password prompt and directory listing. The bottom status bar shows the file is in 'master*' branch, has 0 changes, and is using PowerShell 6.0.

If you have any problems, you can open issues in the [GitHub repo](#).

Understanding file encoding in VS Code and PowerShell

Article • 11/30/2023

When using VS Code to create and edit PowerShell scripts, it's important that your files are saved using the correct character encoding format.

What is file encoding and why is it important?

VS Code manages the interface between a human entering strings of characters into a buffer and reading/writing blocks of bytes to the filesystem. When VS Code saves a file, it uses a text encoding to decide what bytes each character becomes. For more information, see [about_Character_Encoding](#).

Similarly, when PowerShell runs a script it must convert the bytes in a file to characters to reconstruct the file into a PowerShell program. Since VS Code writes the file and PowerShell reads the file, they need to use the same encoding system. This process of parsing a PowerShell script goes: *bytes -> characters -> tokens -> abstract syntax tree -> execution*.

Both VS Code and PowerShell are installed with a sensible default encoding configuration. However, the default encoding used by PowerShell has changed with the release of PowerShell 6. To ensure you have no problems using PowerShell or the PowerShell extension in VS Code, you need to configure your VS Code and PowerShell settings properly.

Common causes of encoding issues

Encoding problems occur when the encoding of VS Code or your script file doesn't match the expected encoding of PowerShell. There is no way for PowerShell to automatically determine the file encoding.

You're more likely to have encoding problems when you're using characters not in the [7-bit ASCII character set](#). For example:

- Extended non-letter characters like em-dash (–), non-breaking space („) or left double quotation mark („)
- Accented latin characters (É, ü)
- Non-latin characters like Cyrillic (д, ц)

- CJK characters (本, 화, ガ)

Common reasons for encoding issues are:

- The encodings of VS Code and PowerShell haven't been changed from their defaults. For PowerShell 5.1 and below, the default encoding is different from VS Code's.
- Another editor has opened and overwritten the file in a new encoding. This often happens with the ISE.
- The file is checked into source control in an encoding that's different from what VS Code or PowerShell expects. This can happen when collaborators use editors with different encoding configurations.

How to tell when you have encoding issues

Often encoding errors present themselves as parse errors in scripts. If you find strange character sequences in your script, this can be the problem. In the example below, an en-dash (-) appears as the characters â€" :

```
Output

Send-MailMessage : A positional parameter cannot be found that accepts
argument 'Testing FuseMail SMTP...'.
At C:\Users\<User>\<OneDrive>\Development\PowerShell\Scripts\Send-
EmailUsingSmtpRelay.ps1:6 char:1
+ Send-MailMessage â&euro;"From $from â&euro;"To $recipient1 â&euro;"Subject
$subject ...
+ ~~~~~
+ CategoryInfo          : InvalidArgument: (:) [Send-MailMessage],
ParameterBindingException
+ FullyQualifiedErrorId :
PositionalParameterNotFound,Microsoft.PowerShell.Commands.SendMailMessage
```

This problem occurs because VS Code encodes the character – in UTF-8 as the bytes `0xE2 0x80 0x93`. When these bytes are decoded as Windows-1252, they're interpreted as the characters â€" .

Some strange character sequences that you might see include:

- â€" instead of – (an en-dash)
- â€" instead of — (an em-dash)
- Ä,,2 instead of Å
- Â instead of ® (a non-breaking space)
- Æ© instead of é

This handy [reference](#) lists the common patterns that indicate a UTF-8/Windows-1252 encoding problem.

How the PowerShell extension in VS Code interacts with encodings

The PowerShell extension interacts with scripts in a number of ways:

1. When scripts are edited in VS Code, the contents are sent by VS Code to the extension. The [Language Server Protocol](#) mandates that this content is transferred in UTF-8. Therefore, it isn't possible for the extension to get the wrong encoding.
2. When scripts are executed directly in the Integrated Console, they're read from the file by PowerShell directly. If PowerShell's encoding differs from VS Code's, something can go wrong here.
3. When a script that's open in VS Code references another script that isn't open in VS Code, the extension falls back to loading that script's content from the file system. The PowerShell extension defaults to UTF-8 encoding, but uses [byte-order mark](#), or BOM, detection to select the correct encoding.

The problem occurs when assuming the encoding of BOM-less formats (like [UTF-8](#) with no BOM and [Windows-1252](#)). The PowerShell extension defaults to UTF-8. The extension can't change VS Code's encoding settings. For more information, see [issue #824](#).

Choosing the right encoding

Different systems and applications can use different encodings:

- In .NET Standard, on the web, and in the Linux world, UTF-8 is now the dominant encoding.
- Many .NET Framework applications use [UTF-16](#). For historical reasons, this is sometimes called "Unicode", a term that now refers to a broad [standard](#) that includes both UTF-8 and UTF-16.
- On Windows, many native applications that predate Unicode continue to use Windows-1252 by default.

Unicode encodings also have the concept of a byte-order mark (BOM). BOMs occur at the beginning of text to tell a decoder which encoding the text is using. For multi-byte encodings, the BOM also indicates [endianness](#) of the encoding. BOMs are designed to

be bytes that rarely occur in non-Unicode text, allowing a reasonable guess that text is Unicode when a BOM is present.

BOMs are optional and their adoption isn't as popular in the Linux world because a dependable convention of UTF-8 is used everywhere. Most Linux applications presume that text input is encoded in UTF-8. While many Linux applications will recognize and correctly handle a BOM, a number don't, leading to artifacts in text manipulated with those applications.

Therefore:

- If you work primarily with Windows applications and Windows PowerShell, you should prefer an encoding like UTF-8 with BOM or UTF-16.
- If you work across platforms, you should prefer UTF-8 with BOM.
- If you work mainly in Linux-associated contexts, you should prefer UTF-8 without BOM.
- Windows-1252 and latin-1 are essentially legacy encodings that you should avoid if possible. However, some older Windows applications may depend on them.
- It's also worth noting that script signing is [encoding-dependent](#), meaning a change of encoding on a signed script will require resigning.

Configuring VS Code

VS Code's default encoding is UTF-8 without BOM.

To set [VS Code's encoding](#), go to the VS Code settings (`ctrl + ,`) and set the `"files.encoding"` setting:

JSON

```
"files.encoding": "utf8bom"
```

Some possible values are:

- `utf8`: [UTF-8] without BOM
- `utf8bom`: [UTF-8] with BOM
- `utf16le`: Little endian [UTF-16]
- `utf16be`: Big endian [UTF-16]
- `windows1252`: [Windows-1252]

You should get a dropdown for this in the GUI view, or completions for it in the JSON view.

You can also add the following to autodetect encoding when possible:

JSON

```
"files.autoGuessEncoding": true
```

If you don't want these settings to affect all file types, VS Code also allows per-language configurations. Create a language-specific setting by putting settings in a [`<language-name>`] field. For example:

JSON

```
[powershell]: {
  "files.encoding": "utf8bom",
  "files.autoGuessEncoding": true
}
```

You may also want to consider installing the [Gremlins tracker](#) for Visual Studio Code. This extension reveals certain Unicode characters that easily corrupted because they're invisible or look like other normal characters.

Configuring PowerShell

PowerShell's default encoding varies depending on version:

- In PowerShell 6+, the default encoding is UTF-8 without BOM on all platforms.
- In Windows PowerShell, the default encoding is usually Windows-1252, which is an extension of [latin-1](#) (also known as ISO 8859-1).

In PowerShell 5+ you can find your default encoding with this:

PowerShell

```
[psobject].Assembly.GetTypes() | Where-Object { $_.Name -eq 'ClrFacade' } |
  ForEach-Object {
    $_.GetMethod('GetDefaultEncoding',
    [System.Reflection.BindingFlags]'nonpublic,static').Invoke($null, @()) }
```

The following [script](#) can be used to determine what encoding your PowerShell session infers for a script without a BOM.

PowerShell

```

$badBytes = [byte[]]@(0xC3, 0x80)
$utf8Str = [System.Text.Encoding]::UTF8.GetString($badBytes)
$bytes = [System.Text.Encoding]::ASCII.GetBytes('Write-Output '') +
[byte[]]@(0xC3, 0x80) + [byte[]]@(0x22)
$path = Join-Path ([System.IO.Path]::GetTempPath()) 'encodingtest.ps1'

try
{
    [System.IO.File]::WriteAllBytes($path, $bytes)

    switch (& $path)
    {
        $utf8Str
        {
            return 'UTF-8'
            break
        }

        default
        {
            return 'Windows-1252'
            break
        }
    }
}
finally
{
    Remove-Item $path
}

```

It's possible to configure PowerShell to use a given encoding more generally using profile settings. See the following articles:

- @mklement0's [answer about PowerShell encoding on Stack Overflow ↗](#).
- @rkeithhill's [blog post about dealing with BOM-less UTF-8 input in PowerShell ↗](#).

It's not possible to force PowerShell to use a specific input encoding. PowerShell 5.1 and below, running on Windows with the locale set to en-US, defaults to Windows-1252 encoding when there's no BOM. Other locale settings may use a different encoding. To ensure interoperability, it's best to save scripts in a Unicode format with a BOM.

Important

Any other tools you have that touch PowerShell scripts may be affected by your encoding choices or re-encode your scripts to another encoding.

Existing scripts

Scripts already on the file system may need to be re-encoded to your new chosen encoding. In the bottom bar of VS Code, you'll see the label UTF-8. Click it to open the action bar and select **Save with encoding**. You can now pick a new encoding for that file. See [VS Code's encoding ↗](#) for full instructions.

If you need to re-encode multiple files, you can use the following script:

PowerShell

```
Get-ChildItem *.ps1 -Recurse | ForEach-Object {
    $content = Get-Content -Path $_
    Set-Content -Path $_.FullName -Value $content -Encoding UTF8 -PassThru -
    Force
}
```

The PowerShell Integrated Scripting Environment (ISE)

If you also edit scripts using the PowerShell ISE, you need to synchronize your encoding settings there.

The ISE should honor a BOM, but it's also possible to use reflection to [set the encoding ↗](#). Note that this wouldn't be persisted between startups.

Source control software

Some source control tools, such as git, ignore encodings; git just tracks the bytes. Others, like Azure DevOps or Mercurial, may not. Even some git-based tools rely on decoding text.

When this is the case, make sure you:

- Configure the text encoding in your source control to match your VS Code configuration.
- Ensure all your files are checked into source control in the relevant encoding.
- Be wary of changes to the encoding received through source control. A key sign of this is a diff indicating changes but where nothing seems to have changed (because bytes have but characters have not).

Collaborators' environments

On top of configuring source control, ensure that your collaborators on any files you share don't have settings that override your encoding by re-encoding PowerShell files.

Other programs

Any other program that reads or writes a PowerShell script may re-encode it.

Some examples are:

- Using the clipboard to copy and paste a script. This is common in scenarios like:
 - Copying a script into a VM
 - Copying a script out of an email or webpage
 - Copying a script into or out of a Microsoft Word or PowerPoint document
- Other text editors, such as:
 - Notepad
 - vim
 - Any other PowerShell script editor
- Text editing utilities, like:
 - `Get-Content` / `Set-Content` / `Out-File`
 - PowerShell redirection operators like `>` and `>>`
 - `sed` / `awk`
- File transfer programs, like:
 - A web browser, when downloading scripts
 - A file share

Some of these tools deal in bytes rather than text, but others offer encoding configurations. In those cases where you need to configure an encoding, you need to make it the same as your editor encoding to prevent problems.

Other resources on encoding in PowerShell

There are a few other nice posts on encoding and configuring encoding in PowerShell that are worth a read:

- [about_Character_Encoding](#)
- [@mklement0's summary of PowerShell encoding on Stack Overflow ↗](#)
- Previous issues opened on VS Code-PowerShell for encoding problems:
 - [#1308 ↗](#)
 - [#1628 ↗](#)
 - [#1680 ↗](#)
 - [#1744 ↗](#)
 - [#1751 ↗](#)
- [The classic Joel on Software write up about Unicode ↗](#)
- [Encoding in .NET Standard ↗](#)

Using Visual Studio Code to debug compiled cmdlets

Article • 02/08/2023

This guide shows you how to interactively debug C# source code for a compiled PowerShell module using Visual Studio Code (VS Code) and the C# extension.

Some familiarity with the Visual Studio Code debugger is assumed.

- For a general introduction to the VS Code debugger, see [Debugging in Visual Studio Code ↗](#).
- For examples of debugging PowerShell script files and modules, see [Using Visual Studio Code for remote editing and debugging](#).

This guide assumes you have read and followed the instructions in the [Writing Portable Modules](#) guide.

Creating a build task

Build your project automatically before launching a debugging session. Rebuilding ensures that you debug the latest version of your code.

Configure a build task:

1. In the **Command Palette**, run the **Configure Default Build Task** command.
[Run Configure Default Build Task](#)
2. In the **Select a task to configure** dialog, choose **Create tasks.json file from template**.
3. In the **Select a Task Template** dialog, choose **.NET Core**.

A new `tasks.json` file is created if one doesn't exist yet.

To test your build task:

1. In the **Command Palette**, run the **Run Build Task** command.
2. In the **Select the build task to run** dialog, choose **build**.

Information about DLL files being locked

By default, a successful build doesn't show output in the terminal pane. If you see output that contains the text **Project file doesn't exist**, you should edit the `tasks.json` file. Include the explicit path to the C# project expressed as `"${workspaceFolder}/myModule"`. In this example, `myModule` is the name of the project folder. This entry must go after the `build` entry in the `args` list as follows:

```
JSON

{
  "label": "build",
  "command": "dotnet",
  "type": "shell",
  "args": [
    "build",
    "${workspaceFolder}/myModule",
    // Ask dotnet build to generate full paths for file names.
    "/property:GenerateFullPaths=true",
    // Do not generate summary otherwise it leads to duplicate
errors in Problems panel
    "/consoleloggerparameters:NoSummary",
  ],
  "group": "build",
  "presentation": {
    "reveal": "silent"
  },
  "problemMatcher": "$msCompile"
}
```

When debugging, your module DLL is imported into the PowerShell session in the VS Code terminal. The DLL becomes locked. The following message is displayed when you run the build task without closing the terminal session:

```
Output

Could not copy "obj\Debug\netstandard2.0\myModule.dll" to
"bin\Debug\netstandard2.0\myModule.dll"`.
```

Terminal sessions must be closed before you rebuild.

Setting up the debugger

To debug the PowerShell cmdlet, you need to set up a custom launch configuration. This configuration is used to:

- Build your source code
- Start PowerShell with your module loaded

- Leave PowerShell open in the terminal pane

When you invoke your cmdlet in the terminal session, the debugger stops at any breakpoints set in your source code.

Configuring launch.json for PowerShell

1. Install the [C# for Visual Studio Code](#) extension
2. In the Debug pane, add a debug configuration
3. In the `Select environment` dialog, choose `.NET Core`
4. The `launch.json` file is opened in the editor. With your cursor inside the `configurations` array, you see the `configuration` picker. If you don't see this list, select **Add Configuration**.
5. To create a default debug configuration, select **Launch .NET Core Console App**:
[Launch .NET Core Console App](#)

6. Edit the `name`, `program`, `args`, and `console` fields as follows:

```
JSON

{
    "name": "PowerShell cmdlets: pwsh",
    "type": "coreclr",
    "request": "launch",
    "preLaunchTask": "build",
    "program": "pwsh",
    "args": [
        "-NoExit",
        "-NoProfile",
        "-Command",
        "Import-Module ${workspaceFolder}/myModule/bin/Debug/netstandard2.0/myModule.dll",
    ],
    "cwd": "${workspaceFolder}",
    "stopAtEntry": false,
    "console": "integratedTerminal"
}
```

The `program` field is used to launch `pwsh` so that the cmdlet being debugged can be run. The `-NoExit` argument prevents the PowerShell session from exiting as soon as the module is imported. The path in the `Import-Module` argument is the default build output path when you've followed the [Writing Portable Modules](#) guide. If you've created a

module manifest (.psd1 file), you should use the path to that instead. The / path separator works on Windows, Linux, and macOS. You must use the integrated terminal to run the PowerShell commands you want to debug.

ⓘ Note

If the debugger doesn't stop at any breakpoints, look in the Visual Studio Code Debug Console for a line that says:

```
Loaded '/path/to/myModule.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.
```

If you see this, add "justMyCode": false to your launch config (at the same level as "console": "integratedTerminal".

Configuring launch.json for Windows PowerShell

This launch configuration works for testing your cmdlets in Windows PowerShell (powershell.exe). Create a second launch configuration with the following changes:

1. name should be PowerShell cmdlets: powershell
2. type should be clr
3. program should be powershell

It should look like this:

JSON

```
{
  "name": "PowerShell cmdlets: powershell",
  "type": "clr",
  "request": "launch",
  "preLaunchTask": "build",
  "program": "powershell",
  "args": [
    "-NoExit",
    "-NoProfile",
    "-Command",
    "Import-Module ${workspaceFolder}/myModule/bin/Debug/netstandard2.0/myModule.dll",
  ],
  "cwd": "${workspaceFolder}",
```

```
        "stopAtEntry": false,  
        "console": "integratedTerminal"  
    }  
}
```

Launching a debugging session

Now everything is ready to begin debugging.

- Place a breakpoint in the source code for the cmdlet you want to debug:
[A breakpoint shows as a red dot in the gutter](#)
- Ensure that the relevant **PowerShell cmdlets** configuration is selected in the configuration drop-down menu in the **Debug** view:
[Select the launch configuration](#)
- Press **F5** or click on the **Start Debugging** button
- Switch to the terminal pane and invoke your cmdlet:

[Invoke the cmdlet](#)

- Execution stops at the breakpoint:

[Execution halts at breakpoint](#)

You can step through the source code, inspect variables, and inspect the call stack.

To end debugging, click **Stop** in the debug toolbar or press **Shift** + **F5**. The shell used for debugging exits and releases the lock on the compiled DLL file.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PowerShell scripting performance considerations

Article • 05/26/2025

PowerShell scripts that leverage .NET directly and avoid the pipeline tend to be faster than idiomatic PowerShell. Idiomatic PowerShell uses cmdlets and PowerShell functions, often leveraging the pipeline, and resorting to .NET only when necessary.

! Note

Many of the techniques described here aren't idiomatic PowerShell and may reduce the readability of a PowerShell script. Script authors are advised to use idiomatic PowerShell unless performance dictates otherwise.

Suppressing output

There are many ways to avoid writing objects to the pipeline.

- Assignment or file redirection to `$null`
- Casting to `[void]`
- Pipe to `out-Null`

The speeds of assigning to `$null`, casting to `[void]`, and file redirection to `$null` are almost identical. However, calling `out-Null` in a large loop can be significantly slower, especially in PowerShell 5.1.

PowerShell

```
$tests = @{
    'Assign to $null' = {
        $arrayList = [System.Collections.ArrayList]::new()
        foreach ($i in 0..$args[0]) {
            $null = $arraylist.Add($i)
        }
    }
    'Cast to [void]' = {
        $arrayList = [System.Collections.ArrayList]::new()
        foreach ($i in 0..$args[0]) {
            [void] $arraylist.Add($i)
        }
    }
    'Redirect to $null' = {
        $arrayList = [System.Collections.ArrayList]::new()
        foreach ($i in 0..$args[0]) {
```

```

        $arraylist.Add($i) > $null
    }
}
'Pipe to Out-Null' = {
    $arrayList = [System.Collections.ArrayList]::new()
    foreach ($i in 0..$args[0]) {
        $arraylist.Add($i) | Out-Null
    }
}
}

10kb, 50kb, 100kb | ForEach-Object {
    $groupResult = foreach ($test in $tests.GetEnumerator()) {
        $ms = (Measure-Command { & $test.Value $_ }).TotalMilliseconds

        [pscustomobject]@{
            Iterations      = $_
            Test           = $test.Key
            TotalMilliseconds = [Math]::Round($ms, 2)
        }

        [GC]::Collect()
        [GC]::WaitForPendingFinalizers()
    }

    $groupResult = $groupResult | Sort-Object TotalMilliseconds
    $groupResult | Select-Object *, @{
        Name      = 'RelativeSpeed'
        Expression = {
            $relativeSpeed = $_.TotalMilliseconds /
                $groupResult[0].TotalMilliseconds
            [Math]::Round($relativeSpeed, 2).ToString() + 'x'
        }
    }
}

```

These tests were run on a Windows 11 machine in PowerShell 7.3.4. The results are shown below:

Output

Iterations	Test	TotalMilliseconds	RelativeSpeed
10240	Assign to \$null	36.74	1x
10240	Redirect to \$null	55.84	1.52x
10240	Cast to [void]	62.96	1.71x
10240	Pipe to Out-Null	81.65	2.22x
51200	Assign to \$null	193.92	1x
51200	Cast to [void]	200.77	1.04x
51200	Redirect to \$null	219.69	1.13x
51200	Pipe to Out-Null	329.62	1.7x
102400	Redirect to \$null	386.08	1x
102400	Assign to \$null	392.13	1.02x

```
102400 Cast to [void]  
102400 Pipe to Out-Null
```

```
405.24 1.05x  
572.94 1.48x
```

The times and relative speeds can vary depending on the hardware, the version of PowerShell, and the current workload on the system.

Array addition

Generating a list of items is often done using an array with the addition operator:

```
PowerShell
```

```
$results = @()  
$results += Get-Something  
$results += Get-SomethingElse  
$results
```

ⓘ Note

In PowerShell 7.5, array addition was optimized and no longer creates a new array for each operation. The performance considerations described here still apply to PowerShell versions prior to 7.5. For more information, see [What's New in PowerShell 7.5](#).

Array addition is inefficient because arrays have a fixed size. Each addition to the array creates a new array big enough to hold all elements of both the left and right operands. The elements of both operands are copied into the new array. For small collections, this overhead may not matter. Performance can suffer for large collections.

There are a couple of alternatives. If you don't actually require an array, instead consider using a typed generic list (`[List<T>]`):

```
PowerShell
```

```
$results = [System.Collections.Generic.List[Object]]::new()  
$results.AddRange((Get-Something))  
$results.AddRange((Get-SomethingElse))  
$results
```

The performance impact of using array addition grows exponentially with the size of the collection and the number additions. This code compares explicitly assigning values to an array with using array addition and using the `Add(T)` method on a `[List<T>]` object. It defines explicit assignment as the baseline for performance.

PowerShell

```
$tests = @{
    'PowerShell Explicit Assignment' = {
        param($Count)

        $result = foreach($i in 1..$Count) {
            $i
        }
    }
    '.Add(T) to List<T>' = {
        param($Count)

        $result = [Collections.Generic.List[int]]::new()
        foreach($i in 1..$Count) {
            $result.Add($i)
        }
    }
    '+= Operator to Array' = {
        param($Count)

        $result = @()
        foreach($i in 1..$Count) {
            $result += $i
        }
    }
}

5kb, 10kb, 100kb | ForEach-Object {
    $groupResult = foreach($test in $tests.GetEnumerator()) {
        $ms = (Measure-Command { & $test.Value -Count $_ }).TotalMilliseconds

        [pscustomobject]@{
            CollectionSize      = $_
            Test                = $test.Key
            TotalMilliseconds = [Math]::Round($ms, 2)
        }
    }

    [GC]::Collect()
    [GC]::WaitForPendingFinalizers()
}

$groupResult = $groupResult | Sort-Object TotalMilliseconds
$groupResult | Select-Object *, @{
    Name      = 'RelativeSpeed'
    Expression = {
        $relativeSpeed = $_.TotalMilliseconds /
        $groupResult[0].TotalMilliseconds
        [Math]::Round($relativeSpeed, 2).ToString() + 'x'
    }
}
```

These tests were run on a Windows 11 machine in PowerShell 7.3.4.

Output

CollectionSize Test	TotalMilliseconds	RelativeSpeed
5120 PowerShell Explicit Assignment	26.65	1x
5120 .Add(T) to List<T>	110.98	4.16x
5120 += Operator to Array	402.91	15.12x
10240 PowerShell Explicit Assignment	0.49	1x
10240 .Add(T) to List<T>	137.67	280.96x
10240 += Operator to Array	1678.13	3424.76x
102400 PowerShell Explicit Assignment	11.18	1x
102400 .Add(T) to List<T>	1384.03	123.8x
102400 += Operator to Array	201991.06	18067.18x

When you're working with large collections, array addition is dramatically slower than adding to a `List<T>`.

When using a `[List<T>]` object, you need to create the list with a specific type, like `[string]` or `[int]`. When you add objects of a different type to the list, they are cast to the specified type. If they can't be cast to the specified type, the method raises an exception.

PowerShell

```
$intList = [System.Collections.Generic.List[int]]::new()
[int]$intList.Add(1)
[int]$intList.Add('2')
[int]$intList.Add(3.0)
[int]$intList.Add('Four')
[int]$intList
```

Output

```
MethodException:
Line |
 5 |     $intList.Add('Four')
   | ~~~~~
   | Cannot convert argument "item", with value: "Four", for "Add" to type
   | "System.Int32": "Cannot convert value "Four" to type "System.Int32".
   | Error: "The input string 'Four' was not in a correct format.""
```

```
1
2
3
```

When you need the list to be a collection of different types of objects, create it with `[Object]` as the list type. You can enumerate the collection inspect the types of the objects in it.

PowerShell

```
$objectList = [System.Collections.Generic.List[Object]]::new()
)objectList.Add(1)
)objectList.Add('2')
)objectList.Add(3.0)
)objectList | ForEach-Object { "$_ is $($_.GetType())" }
```

Output

```
1 is int
2 is string
3 is double
```

If you do require an array, you can call the `ToArray()` method on the list or you can let PowerShell create the array for you:

PowerShell

```
$results = @(
    Get-Something
    Get-SomethingElse
)
```

In this example, PowerShell creates an `[ArrayList]` to hold the results written to the pipeline inside the array expression. Just before assigning to `$results`, PowerShell converts the `[ArrayList]` to an `[Object[]]`.

String addition

Strings are immutable. Each addition to the string actually creates a new string big enough to hold the contents of both the left and right operands, then copies the elements of both operands into the new string. For small strings, this overhead may not matter. For large strings, this can affect performance and memory consumption.

There are at least two alternatives:

- The `-join` operator concatenates strings
- The .NET `[StringBuilder]` class provides a mutable string

The following example compares the performance of these three methods of building a string.

PowerShell

```

$tests = @{
    'StringBuilder' = {
        $sb = [System.Text.StringBuilder]::new()
        foreach ($i in 0..$args[0]) {
            $sb = $sb.AppendLine("Iteration $i")
        }
        $sb.ToString()
    }
    'Join operator' = {
        $string = @(
            foreach ($i in 0..$args[0]) {
                "Iteration $i"
            }
        ) -join "`n"
        $string
    }
    'Addition Assignment +=' = {
        $string = ''
        foreach ($i in 0..$args[0]) {
            $string += "Iteration $i`n"
        }
        $string
    }
}

10kb, 50kb, 100kb | ForEach-Object {
    $groupResult = foreach ($test in $tests.GetEnumerator()) {
        $ms = (Measure-Command { & $test.Value $_ }).TotalMilliseconds

        [pscustomobject]@{
            Iterations      = $_
            Test           = $test.Key
            TotalMilliseconds = [Math]::Round($ms, 2)
        }

        [GC]::Collect()
        [GC]::WaitForPendingFinalizers()
    }

    $groupResult = $groupResult | Sort-Object TotalMilliseconds
    $groupResult | Select-Object *, @{
        Name      = 'RelativeSpeed'
        Expression = {
            $relativeSpeed = $_.TotalMilliseconds /
                $groupResult[0].TotalMilliseconds
            [Math]::Round($relativeSpeed, 2).ToString() + 'x'
        }
    }
}

```

These tests were run on a Windows 11 machine in PowerShell 7.4.2. The output shows that the `-join` operator is the fastest, followed by the `[StringBuilder]` class.

Output

Iterations	Test	TotalMilliseconds	RelativeSpeed
10240	Join operator	14.75	1x
10240	StringBuilder	62.44	4.23x
10240	Addition Assignment +=	619.64	42.01x
51200	Join operator	43.15	1x
51200	StringBuilder	304.32	7.05x
51200	Addition Assignment +=	14225.13	329.67x
102400	Join operator	85.62	1x
102400	StringBuilder	499.12	5.83x
102400	Addition Assignment +=	67640.79	790.01x

The times and relative speeds can vary depending on the hardware, the version of PowerShell, and the current workload on the system.

Processing large files

The idiomatic way to process a file in PowerShell might look something like:

```
PowerShell
```

```
Get-Content $path | Where-Object Length -GT 10
```

This can be an order of magnitude slower than using .NET APIs directly. For example, you can use the .NET `[StreamReader]` class:

```
PowerShell
```

```
try {
    $reader = [System.IO.StreamReader]::new($path)
    while (-not $reader.EndOfStream) {
        $line = $reader.ReadLine()
        if ($line.Length -gt 10) {
            $line
        }
    }
}
finally {
    if ($reader) {
        $reader.Dispose()
    }
}
```

You could also use the `ReadLines` method of `[System.IO.File]`, which wraps `StreamReader`, simplifies the reading process:

```
PowerShell
```

```
foreach ($line in [System.IO.File]::ReadLines($path)) {  
    if ($line.Length -gt 10) {  
        $line  
    }  
}
```

Looking up entries by property in large collections

It's common to need to use a shared property to identify the same record in different collections, like using a name to retrieve an ID from one list and an email from another. Iterating over the first list to find the matching record in the second collection is slow. In particular, the repeated filtering of the second collection has a large overhead.

Given two collections, one with an **Id** and **Name**, the other with **Name** and **Email**:

```
PowerShell
```

```
$Employees = 1..10000 | ForEach-Object {  
    [pscustomobject]@{  
        Id    = $_  
        Name = "Name$_"  
    }  
}  
  
$Accounts = 2500..7500 | ForEach-Object {  
    [pscustomobject]@{  
        Name  = "Name$_"  
        Email = "Name$_@fabrikam.com"  
    }  
}
```

The usual way to reconcile these collections to return a list of objects with the **Id**, **Name**, and **Email** properties might look like this:

```
PowerShell
```

```
$Results = $Employees | ForEach-Object -Process {  
    $Employee = $_  
  
    $Account = $Accounts | Where-Object -FilterScript {  
        $_.Name -eq $Employee.Name  
    }  
  
    [pscustomobject]@{  
        Id    = $Employee.Id  
        Name = $Employee.Name  
    }
```

```
        Email = $Account.Email  
    }  
}
```

However, that implementation has to filter all 5000 items in the `$Accounts` collection once for every item in the `$Employee` collection. That can take minutes, even for this single-value lookup.

Instead, you can make a [Hash Table](#) that uses the shared **Name** property as a key and the matching account as the value.

PowerShell

```
$LookupHash = @{}  
foreach ($Account in $Accounts) {  
    $LookupHash[$Account.Name] = $Account  
}
```

Looking up keys in a hash table is much faster than filtering a collection by property values. Instead of checking every item in the collection, PowerShell can check if the key is defined and use its value.

PowerShell

```
$Results = $Employees | ForEach-Object -Process {  
    $Email = $LookupHash[$_.Name].Email  
    [pscustomobject]@{  
        Id      = $_.Id  
        Name   = $_.Name  
        Email  = $Email  
    }  
}
```

This is much faster. While the looping filter took minutes to complete, the hash lookup takes less than a second.

Use Write-Host carefully

The `Write-Host` command should only be used when you need to write formatted text to the host console, rather than writing objects to the `Success` pipeline.

`Write-Host` can be an order of magnitude slower than `[Console]::WriteLine()` for specific hosts like `pwsh.exe`, `powershell.exe`, or `powershell_ise.exe`. However, `[Console]::WriteLine()` isn't guaranteed to work in all hosts. Also, output written using `[Console]::WriteLine()` doesn't get written to transcripts started by `Start-Transcript`.

JIT compilation

PowerShell compiles the script code to bytecode that's interpreted. Beginning in PowerShell 3, for code that's repeatedly executed in a loop, PowerShell can improve performance by Just-in-time (JIT) compiling the code into native code.

Loops that have fewer than 300 instructions are eligible for JIT-compilation. Loops larger than that are too costly to compile. When the loop has executed 16 times, the script is JIT-compiled in the background. When the JIT-compilation completes, execution is transferred to the compiled code.

Avoid repeated calls to a function

Calling a function can be an expensive operation. If you're calling a function in a long running tight loop, consider moving the loop inside the function.

Consider the following examples:

```
PowerShell

$tests = @{
    'Simple for-loop'      = {
        param([int] $RepeatCount, [random] $RanGen)

        for ($i = 0; $i -lt $RepeatCount; $i++) {
            $null = $RanGen.Next()
        }
    }
    'Wrapped in a function' = {
        param([int] $RepeatCount, [random] $RanGen)

        function Get-RandomNumberCore {
            param ($Rng)

            $Rng.Next()
        }

        for ($i = 0; $i -lt $RepeatCount; $i++) {
            $null = Get-RandomNumberCore -Rng $RanGen
        }
    }
    'for-loop in a function' = {
        param([int] $RepeatCount, [random] $RanGen)

        function Get-RandomNumberAll {
            param ($Rng, $Count)

            for ($i = 0; $i -lt $Count; $i++) {
                $null = $Rng.Next()
            }
        }
    }
}
```

```

        }

    }

    Get-RandomNumberAll -Rng $RanGen -Count $RepeatCount
}

}

5kb, 10kb, 100kb | ForEach-Object {
    $Rng = [random]::new()
    $groupResult = foreach ($test in $tests.GetEnumerator()) {
        $ms = Measure-Command { & $test.Value -RepeatCount $_ -RanGen $Rng }

        [pscustomobject]@{
            CollectionSize      = $_
            Test                = $test.Key
            TotalMilliseconds = [Math]::Round($ms.TotalMilliseconds,2)
        }
    }

    [GC]::Collect()
    [GC]::WaitForPendingFinalizers()
}

$groupResult = $groupResult | Sort-Object TotalMilliseconds
$groupResult | Select-Object *, @{
    Name      = 'RelativeSpeed'
    Expression = {
        $relativeSpeed = $_.TotalMilliseconds /
$groupResult[0].TotalMilliseconds
        [Math]::Round($relativeSpeed, 2).ToString() + 'x'
    }
}
}
}

```

The **Basic for-loop** example is the base line for performance. The second example wraps the random number generator in a function that's called in a tight loop. The third example moves the loop inside the function. The function is only called once but the code still generates the same amount of random numbers. Notice the difference in execution times for each example.

Output

CollectionSize	Test	TotalMilliseconds	RelativeSpeed
5120	for-loop in a function	9.62	1x
5120	Simple for-loop	10.55	1.1x
5120	Wrapped in a function	62.39	6.49x
10240	Simple for-loop	17.79	1x
10240	for-loop in a function	18.48	1.04x
10240	Wrapped in a function	127.39	7.16x
102400	for-loop in a function	179.19	1x
102400	Simple for-loop	181.58	1.01x
102400	Wrapped in a function	1155.57	6.45x

Avoid wrapping cmdlet pipelines

Most cmdlets are implemented for the pipeline, which is a sequential syntax and process. For example:

```
PowerShell  
cmdlet1 | cmdlet2 | cmdlet3
```

Initializing a new pipeline can be expensive, therefore you should avoid wrapping a cmdlet pipeline into another existing pipeline.

Consider the following example. The `Input.csv` file contains 2100 lines. The `Export-Csv` command is wrapped inside the `ForEach-Object` pipeline. The `Export-Csv` cmdlet is invoked for every iteration of the `ForEach-Object` loop.

```
PowerShell  
  
$measure = Measure-Command -Expression {  
    Import-Csv .\Input.csv | ForEach-Object -Begin { $Id = 1 } -Process {  
        [pscustomobject]@{  
            Id      = $Id  
            Name   = $_.opened_by  
        } | Export-Csv .\Output1.csv -Append  
    }  
}  
  
'Wrapped = {0:N2} ms' -f $measure.TotalMilliseconds
```

```
Output  
  
Wrapped = 15,968.78 ms
```

For the next example, the `Export-Csv` command was moved outside of the `ForEach-Object` pipeline. In this case, `Export-Csv` is invoked only once, but still processes all objects passed out of `ForEach-Object`.

```
PowerShell  
  
$measure = Measure-Command -Expression {  
    Import-Csv .\Input.csv | ForEach-Object -Begin { $Id = 2 } -Process {  
        [pscustomobject]@{  
            Id      = $Id  
            Name   = $_.opened_by  
        }  
    } | Export-Csv .\Output2.csv
```

```
}
```

```
'Unwrapped = {0:N2} ms' -f $measure.TotalMilliseconds
```

Output

```
Unwrapped = 42.92 ms
```

The unwrapped example is **372 times faster**. Also, notice that the first implementation requires the **Append** parameter, which isn't required for the later implementation.

Object creation

Creating objects using the `New-Object` cmdlet can be slow. The following code compares the performance of creating objects using the `New-Object` cmdlet to the `[pscustomobject]` type accelerator.

PowerShell

```
Measure-Command {
    $test = 'PSCustomObject'
    for ($i = 0; $i -lt 100000; $i++) {
        $resultObject = [pscustomobject]@{
            Name = 'Name'
            Path = 'FullName'
        }
    }
} | Select-Object @{n='Test';e={$test}},TotalSeconds

Measure-Command {
    $test = 'New-Object'
    for ($i = 0; $i -lt 100000; $i++) {
        $resultObject = New-Object -TypeName psobject -Property @{
            Name = 'Name'
            Path = 'FullName'
        }
    }
} | Select-Object @{n='Test';e={$test}},TotalSeconds
```

Output

Test	TotalSeconds
---	-----
PSCustomObject	0.48
New-Object	3.37

PowerShell 5.0 added the `new()` static method for all .NET types. The following code compares the performance of creating objects using the `New-Object` cmdlet to the `new()` method.

```
PowerShell

Measure-Command {
    $test = 'new() method'
    for ($i = 0; $i -lt 100000; $i++) {
        $sb = [System.Text.StringBuilder]::new(1000)
    }
} | Select-Object @{n='Test';e={$test}},TotalSeconds

Measure-Command {
    $test = 'New-Object'
    for ($i = 0; $i -lt 100000; $i++) {
        $sb = New-Object -TypeName System.Text.StringBuilder -ArgumentList 1000
    }
} | Select-Object @{n='Test';e={$test}},TotalSeconds
```

Output

Test	TotalSeconds
new() method	0.59
New-Object	3.17

Use `OrderedDictionary` to dynamically create new objects

There are situations where we may need to dynamically create objects based on some input, the perhaps most commonly used way to create a new `PSObject` and then add new properties using the `Add-Member` cmdlet. The performance cost for small collections using this technique may be negligible however it can become very noticeable for big collections. In that case, the recommended approach is to use an `[OrderedDictionary]` and then convert it to a `PSObject` using the `[pscustomobject]` type accelerator. For more information, see the *Creating ordered dictionaries* section of [about_Hash_Tables](#).

Assume you have the following API response stored in the variable `$json`.

```
JSON

{
  "tables": [
    {
      "name": "PrimaryResult",
```

```

    "columns": [
        { "name": "Type", "type": "string" },
        { "name": "TenantId", "type": "string" },
        { "name": "count_", "type": "long" }
    ],
    "rows": [
        [ "Usage", "63613592-b6f7-4c3d-a390-22ba13102111", "1" ],
        [ "Usage", "d436f322-a9f4-4aad-9a7d-271fb66001c", "1" ],
        [ "BillingFact", "63613592-b6f7-4c3d-a390-22ba13102111", "1" ],
        [ "BillingFact", "d436f322-a9f4-4aad-9a7d-271fb66001c", "1" ],
        [ "Operation", "63613592-b6f7-4c3d-a390-22ba13102111", "7" ],
        [ "Operation", "d436f322-a9f4-4aad-9a7d-271fb66001c", "5" ]
    ]
}
]
}

```

Now, suppose you want to export this data to a CSV. First you need to create new objects and add the properties and values using the `Add-Member` cmdlet.

PowerShell

```

$data = $json | ConvertFrom-Json
$columns = $data.tables.columns
$result = foreach ($row in $data.tables.rows) {
    $obj = [psobject]::new()
    $index = 0

    foreach ($column in $columns) {
        $obj | Add-Member -MemberType NoteProperty -Name $column.name -Value
$row[$index++]
    }

    $obj
}

```

Using an `OrderedDictionary`, the code can be translated to:

PowerShell

```

$data = $json | ConvertFrom-Json
$columns = $data.tables.columns
$result = foreach ($row in $data.tables.rows) {
    $obj = [ordered]@{}
    $index = 0

    foreach ($column in $columns) {
        $obj[$column.name] = $row[$index++]
    }
}

```

```
[pscustomobject] $obj  
}
```

In both cases the `$result` output would be same:

Output

Type	TenantId	count_
Usage	63613592-b6f7-4c3d-a390-22ba13102111	1
Usage	d436f322-a9f4-4aad-9a7d-271fbf66001c	1
BillingFact	63613592-b6f7-4c3d-a390-22ba13102111	1
BillingFact	d436f322-a9f4-4aad-9a7d-271fbf66001c	1
Operation	63613592-b6f7-4c3d-a390-22ba13102111	7
Operation	d436f322-a9f4-4aad-9a7d-271fbf66001c	5

The latter approach becomes exponentially more efficient as the number of objects and member properties increases.

Here is a performance comparison of three techniques for creating objects with 5 properties:

PowerShell

```
$tests = @{  
    '[ordered] into [pscustomobject] cast' = {  
        param([int] $Iterations, [string[]] $Props)  
  
        foreach ($i in 1..$Iterations) {  
            $obj = [ordered]@{}  
            foreach ($prop in $Props) {  
                $obj[$prop] = $i  
            }  
            [pscustomobject] $obj  
        }  
    }  
    'Add-Member' = {  
        param([int] $Iterations, [string[]] $Props)  
  
        foreach ($i in 1..$Iterations) {  
            $obj = [psobject]::new()  
            foreach ($prop in $Props) {  
                $obj | Add-Member -MemberType NoteProperty -Name $prop -Value $i  
            }  
            $obj  
        }  
    }  
    'PSObject.Properties.Add' = {  
        param([int] $Iterations, [string[]] $Props)  
  
        # this is how, behind the scenes, `Add-Member` attaches  
        # new properties to our PSObject.  
    }  
}
```

```

# Worth having it here for performance comparison

foreach ($i in 1..$Iterations) {
    $obj = [psobject]::new()
    foreach ($prop in $Props) {
        $obj.psobject.Properties.Add(
            [psnoteproperty]::new($prop, $i))
    }
    $obj
}

$properties = 'Prop1', 'Prop2', 'Prop3', 'Prop4', 'Prop5'

1kb, 10kb, 100kb | ForEach-Object {
    $groupResult = foreach ($test in $tests.GetEnumerator()) {
        $ms = Measure-Command { & $test.Value -Iterations $_ -Props $properties }

        [pscustomobject]@{
            Iterations      = $_
            Test           = $test.Key
            TotalMilliseconds = [Math]::Round($ms.TotalMilliseconds, 2)
        }

        [GC]::Collect()
        [GC]::WaitForPendingFinalizers()
    }

    $groupResult = $groupResult | Sort-Object TotalMilliseconds
    $groupResult | Select-Object *, @{
        Name      = 'RelativeSpeed'
        Expression = {
            $relativeSpeed = $_.TotalMilliseconds /
                $groupResult[0].TotalMilliseconds
            [Math]::Round($relativeSpeed, 2).ToString() + 'x'
        }
    }
}

```

And these are the results:

Output

Iterations	Test	TotalMilliseconds	RelativeSpeed
1024	[ordered] into [pscustomobject] cast	22.00	1x
1024	PSObject.Properties.Add	153.17	6.96x
1024	Add-Member	261.96	11.91x
10240	[ordered] into [pscustomobject] cast	65.24	1x
10240	PSObject.Properties.Add	1293.07	19.82x
10240	Add-Member	2203.03	33.77x
102400	[ordered] into [pscustomobject] cast	639.83	1x

102400 PSObject.Properties.Add
102400 Add-Member

13914.67 21.75x
23496.08 36.72x

Related links

- [\\$null](#)
- [\[void\]](#)
- [Out-Null](#)
- [List<T>](#)
- [Add\(T\) method](#)
- [\[string\]](#)
- [\[int\]](#)
- [\[Object\]](#)
- [ToArray\(\) method](#)
- [\[ArrayList\]](#)
- [\[StringBuilder\]](#)
- [\[StreamReader\]](#)
- [\[File\]::ReadLines\(\) method](#)
- [Write-Host](#)
- [Add-Member](#)

PowerShell module authoring considerations

Article • 11/17/2022

This document includes some guidelines related to how a module is authored for best performance.

Module Manifest Authoring

A module manifest that doesn't use the following guidelines can have a noticeable impact on general PowerShell performance even if the module isn't used in a session.

Command auto-discovery analyzes each module to determine which commands the module exports and this analysis can be expensive. The results of module analysis are cached per user, but the cache isn't available on first run, which is a typical scenario with containers. During module analysis, if the exported commands can be fully determined from the manifest, more expensive analysis of the module can be avoided.

Guidelines

- In the module manifest, don't use wildcards in the `AliasesToExport`, `CmdletsToExport`, and `FunctionsToExport` entries.
- If the module doesn't export commands of a particular type, specify this explicitly in the manifest by specifying `@()`. A missing or `$null` entry is equivalent to specifying the wildcard `*`.

The following should be avoided where possible:

```
PowerShell

@{
    FunctionsToExport = '*'

    # Also avoid omitting an entry, it's equivalent to using a wildcard
    # CmdletsToExport = '*'
    # AliasesToExport = '*'
}
```

Instead, use:

PowerShell

```
@{
    FunctionsToExport = 'Format-Hex', 'Format-Octal'
    CmdletsToExport = @() # Specify an empty array, not $null
    AliasesToExport = @() # Also ensure all three entries are present
}
```

Avoid CDXML

When deciding how to implement your module, there are three primary choices:

- Binary (usually C#)
- Script (PowerShell)
- CDXML (an XML file wrapping CIM)

If the speed of loading your module is important, CDXML is roughly an order of magnitude slower than a binary module.

A binary module loads the fastest because it's compiled ahead of time and can use NGen to JIT compile once per machine.

A script module typically loads a bit more slowly than a binary module because PowerShell must parse the script before compiling and executing it.

A CDXML module is typically much slower than a script module because it must first parse an XML file which then generates quite a bit of PowerShell script that's then parsed and compiled.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Optimize performance using parallel execution

Article • 05/19/2025

PowerShell provides several options for the creation of parallel invocations.

- `Start-Job` runs each job in a separate process, each with a new instance of PowerShell. In many cases, a linear loop is faster. Also, serialization and deserialization can limit the usefulness of the objects returned. This command is built in to all versions of PowerShell.
- `Start-ThreadJob` is a cmdlet found in the **ThreadJob** module. This command uses PowerShell runspaces to create and manage thread-based jobs. These jobs are lighter-weight than the jobs created by `Start-Job` and avoid potential loss of type fidelity required by cross-process serialization and deserialization. The **ThreadJob** module comes with PowerShell 7 and higher. For Windows PowerShell 5.1, you can install this module from the PowerShell Gallery.
- Use the **System.Management.Automation.Runspaces** namespace from the PowerShell SDK to create your own parallel logic. Both `ForEach-Object -Parallel` and `Start-ThreadJob` use PowerShell runspaces to execute the code in parallel.
- Workflows are a feature of Windows PowerShell 5.1. Workflows aren't available in PowerShell 7.0 and higher. Workflows are a special type of PowerShell script that can run in parallel. They're designed for long-running tasks and can be paused and resumed. Workflows aren't recommended for new development. For more information, see [about_Workflows](#).
- `ForEach-Object -Parallel` is a feature of PowerShell 7.0 and higher. Like `Start-ThreadJob`, it uses PowerShell runspaces to create and manage thread-based jobs. This command is designed for use in a pipeline.

Limit execution concurrency

Running scripts in parallel doesn't guarantee improved performance. For example, the following scenarios can benefit from parallel execution:

- Compute intensive scripts on multi-threaded multi-core processors
- Scripts that spend time waiting for results or doing file operations, as long as those operations don't block each other.

It's important to balance the overhead of parallel execution with the type of work done. Also, there are limits to the number of invocations that can run in parallel.

The `Start-ThreadJob` and `ForEach-Object -Parallel` commands have a `ThrottleLimit` parameter to limit the number of jobs running at one time. As more jobs are started, they're queued and wait until the current number of jobs drops below the throttle limit. As of PowerShell 7.1, `ForEach-Object -Parallel` reuses runspaces from a runspace pool by default. The `ThrottleLimit` parameter sets the runspace pool size. The default runspace pool size is 5. You can still create a new runspace for each iteration using the `UseNewRunspace` switch.

The `Start-Job` command doesn't have a `ThrottleLimit` parameter. You have to manage the number of jobs running at one time.

Measure performance

The following function, `Measure-Parallel`, compares the speed of the following parallel execution approaches:

- `Start-Job` - creates a child PowerShell process behind the scenes
- `Start-ThreadJob` - runs each job in a separate thread
- `ForEach-Object -Parallel` - runs each job in a separate thread
- `Start-Process` - invokes an external program asynchronously

ⓘ Note

This approach only makes sense if your parallel tasks only consist of a single call to an external program, as opposed to running a block of PowerShell code. Also, the only way to capture output with this approach is by redirecting to a file.

PowerShell

```
function Measure-Parallel {
    [CmdletBinding()]
    param(
        [ValidateRange(2, 2147483647)]
        [int] $BatchSize = 5,
        [ValidateSet('Job', 'ThreadJob', 'Process', 'ForEachParallel', 'All')]
        [string[]] $Approach,
        # pass a higher count to run multiple batches
        [ValidateRange(2, 2147483647)]
        [int] $JobCount = $BatchSize
    )
```

```

$noForEachParallel = $PSVersionTable.PSVersion.Major -lt 7
$noStartThreadJob = -not (Get-Command -ErrorAction Ignore Start-ThreadJob)

# Translate the approach arguments into their corresponding hashtable keys
# (see below).
if ('All' -eq $Approach) { $Approach = 'Job', 'ThreadJob', 'Process',
'ForEachParallel' }
$approaches = $Approach.ForEach({
    if ($_. -eq 'ForEachParallel') { 'ForEach-Object -Parallel' }
    else { $_ -replace '^', 'Start-' }
})

if ($noStartThreadJob) {
    if ($interactive -or $approaches -contains 'Start-ThreadJob') {
        Write-Warning "Start-ThreadJob is not installed, omitting its test."
        $approaches = $approaches.Where({ $_ -ne 'Start-ThreadJob' })
    }
}
if ($noForEachParallel) {
    if ($interactive -or $approaches -contains 'ForEach-Object -Parallel') {
        Write-Warning 'ForEach-Object -Parallel require PowerShell v7+, omitting its test.'
        $approaches = $approaches.Where({ $_ -ne 'ForEach-Object -Parallel' })
    }
}

# Simulated input: Create 'f0.zip', 'f1.zip', ... file names.
$zipFiles = 0..($JobCount - 1) -replace '^', 'f' -replace '$', '.zip'

# Sample executables to run - here, the native shell is called to simply
# echo the argument given.
$exe = if ($env:OS -eq 'Windows_NT') { 'cmd.exe' } else { 'sh' }

# The list of its arguments *as a single string* - use '{0}' as the
placeholder
# for where the input object should go.
$exeArgList = if ($env:OS -eq 'Windows_NT') {
    '/c "echo {0} > NUL"'
} else {
    '-c "echo {0} > /dev/null"'
}

# A hashtable with script blocks that implement the 3 approaches to
parallelism.
$approachImpl = [ordered] @{

    # child-process-based job
    $approachImpl['Start-Job'] = {
        param([array] $batch)
        $batch |
            ForEach-Object {
                Start-Job {
                    Invoke-Expression ($using:exe + ' ' + ($using:exeArgList -f
$args[0]))
                } -ArgumentList $_
            }
    }
}

```

```

    } |
    Receive-Job -Wait -AutoRemoveJob | Out-Null
}

# thread-based job - requires the ThreadJob module
if (-not $noStartThreadJob) {
    # If Start-ThreadJob is available, add an approach for it.
    $approachImpl['Start-ThreadJob'] = {
        param([array] $batch)
        $batch |
        ForEach-Object {
            Start-ThreadJob -ThrottleLimit $BatchSize {
                Invoke-Expression ($using:exe + ' ' + ($using:exeArgList -
f $args[0]))
            } -ArgumentList $_
        } |
        Receive-Job -Wait -AutoRemoveJob | Out-Null
    }
}

# ForEach-Object -Parallel job
if (-not $noForEachParallel) {
    $approachImpl['ForEach-Object -Parallel'] = {
        param([array] $batch)
        $batch | ForEach-Object -ThrottleLimit $BatchSize -Parallel {
            Invoke-Expression ($using:exe + ' ' + ($using:exeArgList -f $_))
        }
    }
}

# direct execution of an external program
$approachImpl['Start-Process'] = {
    param([array] $batch)
    $batch |
    ForEach-Object {
        Start-Process -NoNewWindow -PassThru $exe -ArgumentList
($exeArgList -f $_)
    } |
    Wait-Process
}

# Partition the array of all indices into subarrays (batches)
$batches = @(
    0..([math]::Ceiling($zipFiles.Count / $batchSize) - 1) | ForEach-Object {
        , $zipFiles[( $_ * $batchSize)..($_ * $batchSize + $batchSize - 1)]
    }
)

$tsTotals = foreach ($appr in $approaches) {
    $i = 0
    $tsTotal = [timespan] 0
    $batches | ForEach-Object {
        Write-Verbose "$batchSize-element '$appr' batch"
        $ts = Measure-Command { & $approachImpl[$appr] $_ | Out-Null }
        $tsTotal += $ts
    }
}

```

```

        if (++$i -eq $batches.Count) {
            # last batch processed.
            if ($batches.Count -gt 1) {
                Write-Verbose ("'$app' processing $JobCount items finished in
" +
                            "$($tsTotal.TotalSeconds.ToString('N2')) secs.")
            }
            $tsTotal # output the overall timing for this approach
        }
    }

# Output a result object with the overall timings.
$oht = [ordered] @{}
$oht['JobCount'] = $JobCount
$oht['BatchSize'] = $BatchSize
$oht['BatchCount'] = $batches.Count
$i = 0
foreach ($app in $approaches) {
    $oht[($app + ' (secs.)')] = $tsTotals[$i++].ToString('N2')
}
[pscustomobject] $oht
}

```

The following example uses `Measure-Parallel` to run 20 jobs in parallel, 5 at a time, using all available approaches.

PowerShell

```
Measure-Parallel -Approach All -BatchSize 5 -JobCount 20 -Verbose
```

The following output comes from a Windows computer running PowerShell 7.5.1. Your timing can vary based on many factors, but the ratios should provide a sense of relative performance.

Output

```

VERBOSE: 5-element 'Start-Job' batch
VERBOSE: 5-element 'Start-Job' batch
VERBOSE: 5-element 'Start-Job' batch
VERBOSE: 5-element 'Start-Job' batch
VERBOSE: 'Start-Job' processing 20 items finished in 7.58 secs.
VERBOSE: 5-element 'Start-ThreadJob' batch
VERBOSE: 5-element 'Start-ThreadJob' batch
VERBOSE: 5-element 'Start-ThreadJob' batch
VERBOSE: 5-element 'Start-ThreadJob' batch
VERBOSE: 'Start-ThreadJob' processing 20 items finished in 2.37 secs.
VERBOSE: 5-element 'Start-Process' batch
VERBOSE: 5-element 'Start-Process' batch
VERBOSE: 5-element 'Start-Process' batch
VERBOSE: 5-element 'Start-Process' batch
VERBOSE: 'Start-Process' processing 20 items finished in 0.26 secs.

```

```
VERBOSE: 5-element 'ForEach-Object -Parallel' batch
VERBOSE: 'ForEach-Object -Parallel' processing 20 items finished in 0.79 secs.
```

```
JobCount : 20
BatchSize : 5
BatchCount : 4
Start-Job (secs.) : 7.58
Start-ThreadJob (secs.) : 2.37
Start-Process (secs.) : 0.26
ForEach-Object -Parallel (secs.) : 0.79
```

Conclusions

- The `Start-Process` approach performs best because it doesn't have the overhead of job management. However, as previously noted, this approach has fundamental limitations.
- The `ForEach-Object -Parallel` adds the least overhead, followed by `Start-ThreadJob`.
- `Start-Job` has the most overhead because of the hidden PowerShell instances it creates for each job.

Acknowledgments

Much of the information in this article is based on the answers from [Santiago Squarzon](#) and [mklement0](#) in this [Stack Overflow post](#).

You may also be interested in the [PSParallelPipeline](#) module created by Santiago Squarzon.

Further reading

- [Start-Job](#)
- [about_Jobs](#)
- [Start-ThreadJob](#)
- [ForEach-Object](#)
- [Start-Process](#)

Portable Modules

Article • 06/29/2023

Windows PowerShell is written for [.NET Framework](#) while PowerShell Core is written for [.NET Core](#). Portable modules are modules that work in both Windows PowerShell and PowerShell Core. While .NET Framework and .NET Core are highly compatible, there are differences in the available APIs between the two. There are also differences in the APIs available in Windows PowerShell and PowerShell Core. Modules intended to be used in both environments need to be aware of these differences.

Porting an existing module

Porting a PSSnapIn

PowerShell [SnapIns](#) aren't supported in PowerShell Core. However, it's trivial to convert a PSSnapIn to a PowerShell module. Typically, the PSSnapIn registration code is in a single source file of a class that derives from [PSSnapIn](#). Remove this source file from the build; it's no longer needed.

Use [New-ModuleManifest](#) to create a new module manifest that replaces the need for the PSSnapIn registration code. Some values from the [PSSnapIn](#) (such as [Description](#)) can be reused within the module manifest.

The [RootModule](#) property in the module manifest should be set to the name of the assembly (`.dll`) implementing the cmdlets.

The .NET Portability Analyzer (aka APIPort)

To port modules written for Windows PowerShell to work with PowerShell Core, start with the [.NET Portability Analyzer](#). Run this tool against your compiled assembly to determine if the .NET APIs used in the module are compatible with .NET Framework, .NET Core, and other .NET runtimes. The tool suggests alternate APIs if they exist. Otherwise, you may need to add [runtime checks](#) and restrict capabilities not available in specific runtimes.

Creating a new module

If creating a new module, the recommendation is to use the [.NET CLI](#).

Installing the PowerShell Standard module template

Once the .NET CLI is installed, install a template library to generate a simple PowerShell module. The module will be compatible with Windows PowerShell, PowerShell Core, Windows, Linux, and macOS.

The following example shows how to install the template:

PowerShell

```
dotnet new install Microsoft.PowerShell.Standard.Module.Template
```

Output

```
The following template packages will be installed:  
  Microsoft.PowerShell.Standard.Module.Template
```

```
Success: Microsoft.PowerShell.Standard.Module.Template::0.1.3 installed the  
following templates:
```

Template Name	Short Name	Language	Tags
PowerShell Standard Module	psmodule	[C#]	Library/PowerShell/Module

Creating a new module project

After the template is installed, you can create a new PowerShell module project using that template. In this example, the sample module is called 'myModule'.

```
PS> mkdir myModule
```

```
Directory: C:\Users\Steve
```

Mode	LastWriteTime	Length	Name
d----	8/3/2018 2:41 PM		myModule

```
PS> cd myModule
```

```
PS C:\Users\Steve\myModule> dotnet new psmodule
```

```
The template "PowerShell Standard Module" was created successfully.
```

```
Processing post-creation actions...
```

```
Restoring C:\Users\Steve\myModule\myModule.csproj:
```

```
  Determining projects to restore...
```

```
  Restored C:\Users\Steve\myModule\myModule.csproj (in 184 ms).
```

```
Restore succeeded.
```

Building the module

Use standard .NET CLI commands to build the project.

PowerShell

```
dotnet build
```

Output

```
PS C:\Users\Steve\myModule> dotnet build
MSBuild version 17.6.3+07e294721 for .NET
  Determining projects to restore...
  All projects are up-to-date for restore.
  PowerShellPG ->
C:\Users\Steve\myModule\bin\Debug\netstandard2.0\myModule.dll

Build succeeded.
  0 Warning(s)
  0 Error(s)

Time Elapsed 00:00:02.36
```

Testing the module

After building the module, you can import it and execute the sample cmdlet.

PowerShell

```
Import-Module .\bin\Debug\netstandard2.0\myModule.dll
Test-SampleCmdlet -?
Test-SampleCmdlet -FavoriteNumber 7 -FavoritePet Cat
```

Output

NAME

Test-SampleCmdlet

SYNTAX

```
Test-SampleCmdlet [-FavoriteNumber] <int> [[-FavoritePet] {Cat | Dog | Horse}] [<CommonParameters>]
```

ALIASES

None

REMARKS

None

```
FavoriteNumber FavoritePet
-----
7 Cat
```

Debugging the module

For a guide on setting up Visual Studio Code to debug the module, see [Using Visual Studio Code for debugging compiled cmdlets](#).

Supporting technologies

The following sections describe in detail some of the technologies used by this template.

.NET Standard Library

[.NET Standard](#) is a formal specification of .NET APIs that are available in all .NET implementations. Managed code targeting .NET Standard works with the .NET Framework and .NET Core versions that are compatible with that version of the .NET Standard.

Note

Although an API may exist in .NET Standard, the API implementation in .NET Core may throw a `PlatformNotSupportedException` at runtime, so to verify compatibility with Windows PowerShell and PowerShell Core, the best practice is to run tests for your module within both environments. Also run tests on Linux and macOS if your module is intended to be cross-platform.

Targeting .NET Standard helps ensure that, as the module evolves, incompatible APIs don't accidentally get introduced into the module. Incompatibilities are discovered at compile time instead of runtime.

However, it isn't required to target .NET Standard for a module to work with both Windows PowerShell and PowerShell Core, as long as you use compatible APIs. The Intermediate Language (IL) is compatible between the two runtimes. You can target .NET Framework 4.6.1, which is compatible with .NET Standard 2.0. If you don't use APIs

outside of .NET Standard 2.0, then your module works with PowerShell Core 6 without recompilation.

PowerShell Standard Library

The [PowerShell Standard](#) library is a formal specification of PowerShell APIs available in all PowerShell versions greater than or equal to the version of that standard.

For example, [PowerShell Standard 5.1](#) is compatible with both Windows PowerShell 5.1 and PowerShell Core 6.0 or newer.

We recommend you compile your module using PowerShell Standard Library. The library ensures the APIs are available and implemented in both Windows PowerShell and PowerShell Core 6. PowerShell Standard is intended to always be forwards-compatible. A module built using PowerShell Standard Library 5.1 will always be compatible with future versions of PowerShell.

Module Manifest

Indicating Compatibility With Windows PowerShell and PowerShell Core

After validating that your module works with both Windows PowerShell and PowerShell Core, the module manifest should explicitly indicate compatibility by using the [CompatiblePSEditions](#) property. A value of `Desktop` means that the module is compatible with Windows PowerShell, while a value of `Core` means that the module is compatible with PowerShell Core. Including both `Desktop` and `Core` means that the module is compatible with both Windows PowerShell and PowerShell Core.

ⓘ Note

`Core` doesn't automatically mean that the module is compatible with Windows, Linux, and macOS. The [CompatiblePSEditions](#) property was introduced in PowerShell v5. Module manifests that use the [CompatiblePSEditions](#) property fail to load in versions prior to PowerShell v5.

Indicating OS Compatibility

First, validate that your module works on Linux and macOS. Next, indicate compatibility with those operating systems in the module manifest. This makes it easier for users to

find your module for their operating system when published to the [PowerShell Gallery](#).

Within the module manifest, the `PrivateData` property has a `PSData` sub-property. The optional `Tags` property of `PSData` takes an array of values that show up in PowerShell Gallery. The PowerShell Gallery supports the following compatibility values:

[\[+\] Expand table](#)

Tag	Description
PSEdition_Core	Compatible with PowerShell Core 6
PSEdition/Desktop	Compatible with Windows PowerShell
Windows	Compatible with Windows
Linux	Compatible with Linux (no specific distro)
macOS	Compatible with macOS

Example:

```
PowerShell

@{
    GUID = "4ae9fd46-338a-459c-8186-07f910774cb8"
    Author = "Microsoft Corporation"
    CompanyName = "Microsoft Corporation"
    Copyright = "(C) Microsoft Corporation. All rights reserved."
    HelpInfoUri = "https://go.microsoft.com/fwlink/?linkid=855962"
    ModuleVersion = "1.2.4"
    PowerShellVersion = "3.0"
    ClrVersion = "4.0"
    RootModule = "PackageManagement.psm1"
    Description = 'PackageManagement (a.k.a. OneGet) is a new way to
discover and install software packages from around the web.
it's a manager or multiplexer of existing package managers (also called
package providers) that unifies Windows package management with a single
Windows PowerShell interface. With PackageManagement, you can do the
following.
- Manage a list of software repositories in which packages can be
searched, acquired and installed
- Discover software packages
- Seamlessly install, uninstall, and inventory packages from one or more
software repositories'

    CmdletsToExport = @(
        'Find-Package',
        'Get-Package',
        'Get-PackageProvider',
        'Install-Package',
        'Uninstall-Package',
        'Update-Package'
    )
}
```

```

'Get-PackageSource',
'Install-Package',
'Import-PackageProvider',
'Find-PackageProvider',
'Install-PackageProvider',
'Register-PackageSource',
'Set-PackageSource',
'Unregister-PackageSource',
'Uninstall-Package',
'Save-Package'
)

FormatsToProcess = @('PackageManagement.format.ps1xml')

PrivateData = @{
    PSData = @{
        Tags = @('PackageManagement', 'PSEdition_Core',
        'PSEdition/Desktop', 'Windows', 'Linux', 'macOS')
        ProjectUri = 'https://oneget.org'
    }
}
}

```

Dependency on Native Libraries

Modules intended for use across different operating systems or processor architectures may depend on a managed library that itself depends on some native libraries.

Prior to PowerShell 7, one would have to have custom code to load the appropriate native dll so that the managed library can find it correctly.

With PowerShell 7, native binaries to load are searched in sub-folders within the managed library's location following a subset of the [.NET RID Catalog](#) notation.

```

managed.dll folder
|
|--- 'win-x64' folder
|     |--- native.dll
|
|--- 'win-x86' folder
|     |--- native.dll
|
|--- 'win-arm' folder
|     |--- native.dll
|
|--- 'win-arm64' folder
|     |--- native.dll
|

```

```
|--- 'linux-x64' folder
|     |--- native.so

|--- 'linux-x86' folder
|     |--- native.so

|--- 'linux-arm' folder
|     |--- native.so

|--- 'linux-arm64' folder
|     |--- native.so

|--- 'osx-x64' folder
|     |--- native.dylib
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to create a Standard Library binary module

Article • 02/02/2023

I recently had an idea for module that I wanted to implement as a binary module. I have yet to create one using the [PowerShell Standard Library](#) so this felt like a good opportunity. I used the [Creating a cross-platform binary module](#) guide to create this module without any roadblocks. We're going to walk that same process and I'll add a little extra commentary along the way.

ⓘ Note

The [original version](#) of this article appeared on the blog written by [@KevinMarquette](#). The PowerShell team thanks Kevin for sharing this content with us. Please check out his blog at [PowerShellExplained.com](#).

What's the PowerShell Standard Library?

The PowerShell Standard Library allows us to create cross platform modules that work in both PowerShell and Windows PowerShell 5.1.

Why binary modules?

When you are writing a module in C# you give up easy access to PowerShell cmdlets and functions. But if you are creating a module that doesn't depend on a lot of other PowerShell commands, the performance benefit can be significant. PowerShell was optimized for the administrator, not the computer. By switching to C#, you get to shed the overhead added by PowerShell.

For example, we have a critical process that does a lot of work with JSON and hashtables. We optimized the PowerShell as much as we could but the process still takes 12 minutes to complete. The module already contained a lot of C# style PowerShell. This makes conversion to a binary module clean and simple. By converting to a binary module, we reduced the process time from over 12 minutes to under four minutes.

Hybrid modules

You can mix binary cmdlets with PowerShell advanced functions. Everything you know about script modules applies the same way. The empty `psm1` file is included so you can add other PowerShell functions later.

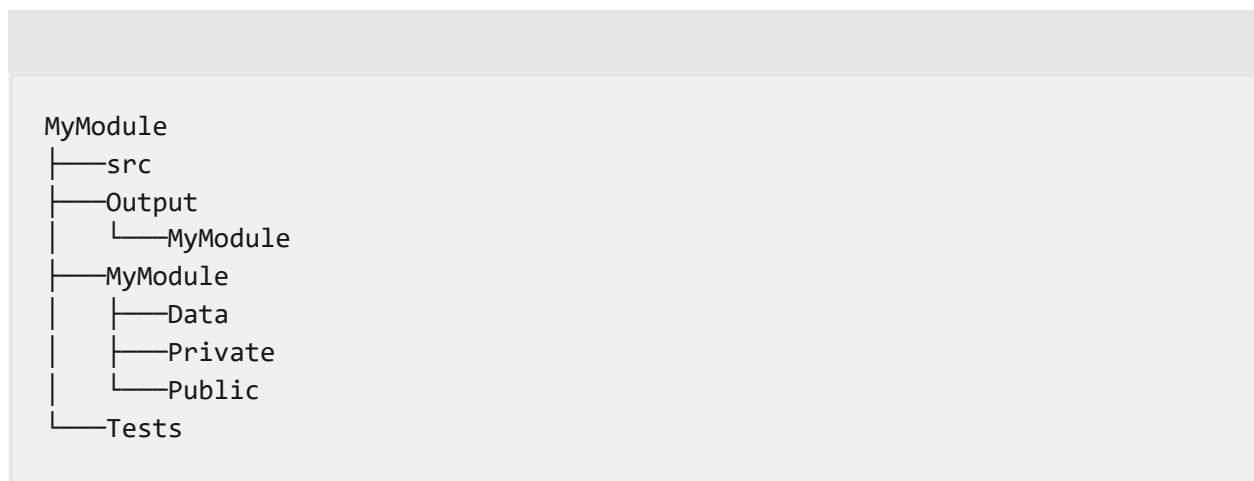
Almost all of the compiled cmdlets that I have created started out as PowerShell functions first. All of our binary modules are really hybrid modules.

Build scripts

I kept the build script simple here. I generally use a large `Invoke-Build` script as part of my CI/CD pipeline. It does more magic like running Pester tests, running `PSScriptAnalyzer`, managing versioning, and publishing to the PSGallery. Once I started using a build script for my modules, I was able to find lots of things to add to it.

Planning the module

The plan for this module is to create a `src` folder for the C# code and structure the rest like I would for a script module. This includes using a build script to compile everything into an `Output` folder. The folder structure looks like this:



Getting Started

First I need to create the folder and create the git repo. I'm using `$module` as a placeholder for the module name. This should make it easier for you to reuse these examples if needed.

```
PowerShell

$module = 'MyModule'
New-Item -Path $module -Type Directory
```

```
Set-Location $module  
git init
```

Then create the root level folders.

PowerShell

```
New-Item -Path 'src' -Type Directory  
New-Item -Path 'Output' -Type Directory  
New-Item -Path 'Tests' -Type Directory  
New-Item -Path $module -Type Directory
```

Binary module setup

This article is focused on the binary module so that's where we'll start. This section pulls examples from the [Creating a cross-platform binary module](#) guide. Review that guide if you need more details or have any issues.

First thing we want to do is check the version of the [dotnet core SDK](#) that we have installed. I'm using 2.1.4, but you should have 2.0.0 or newer before continuing.

PowerShell

```
PS> dotnet --version  
2.1.4
```

I'm working out of the `src` folder for this section.

PowerShell

```
Set-Location 'src'
```

Using the dotnet command, create a new class library.

PowerShell

```
dotnet new classlib --name $module
```

This created the library project in a subfolder but I don't want that extra level of nesting. I'm going to move those files up a level.

PowerShell

```
Move-Item -Path .\$module\* -Destination .\  
Remove-Item $module -Recurse
```

Set the .NET core SDK version for the project. I have the 2.1 SDK so I'm going to specify `2.1.0`. Use `2.0.0` if you're using the 2.0 SDK.

PowerShell

```
dotnet new globaljson --sdk-version 2.1.0
```

Add the **PowerShell Standard Library** [NuGet package](#) to the project. Make sure you use the most recent version available for the level of compatibility that you need. I would default to the latest version but I don't think this module leverages any features newer than PowerShell 3.0.

PowerShell

```
dotnet add package PowerShellStandard.Library --version 7.0.0-preview.1
```

We should have a src folder that looks like this:

PowerShell

```
PS> Get-ChildItem  
Directory: \MyModule\src  


| Mode  | LastWriteTime      | Length | Name            |
|-------|--------------------|--------|-----------------|
| -     | -----              | -----  | -----           |
| d---- | 7/14/2018 9:51 PM  |        | obj             |
| -a--- | 7/14/2018 9:51 PM  | 86     | Class1.cs       |
| -a--- | 7/14/2018 10:03 PM | 259    | MyModule.csproj |
| -a--- | 7/14/2018 10:05 PM | 45     | global.json     |


```

Now we're ready to add our own code to the project.

Building a binary cmdlet

We need to update the `src\Class1.cs` to contain this starter cmdlet:

C#

```
using System;  
using System.Management.Automation;  
  
namespace MyModule
```

```
{
    [Cmdlet( VerbsDiagnostic.Resolve , "MyCmdlet")]
    public class ResolveMyCmdletCommand : PSCmdlet
    {
        [Parameter(Position=0)]
        public Object InputObject { get; set; }

        protected override void EndProcessing()
        {
            this.WriteObject(this.InputObject);
            base.EndProcessing();
        }
    }
}
```

Rename the file to match the class name.

PowerShell

```
Rename-Item .\Class1.cs .\ResolveMyCmdletCommand.cs
```

Then we can build our module.

PowerShell

```
PS> dotnet build

Microsoft (R) Build Engine version 15.5.180.51428 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 18.19 ms for C:\workspace\MyModule\src\MyModule.csproj.
MyModule -> C:\workspace\MyModule\src\bin\Debug\netstandard2.0\MyModule.dll

Build succeeded.
  0 Warning(s)
  0 Error(s)

Time Elapsed 00:00:02.19
```

We can call `Import-Module` on the new dll to load our new cmdlet.

PowerShell

```
PS> Import-Module .\bin\Debug\netstandard2.0\$module.dll
PS> Get-Command -Module $module

 CommandType Name                Version Source
 ----- ----
 Cmdlet      Resolve-MyCmdlet   1.0.0.0 MyModule
```

If the import fails on your system, try updating .NET to 4.7.1 or newer. The [Creating a cross-platform binary module](#) guide goes into more details on .NET support and compatibility for older versions of .NET.

Module manifest

It's cool that we can import the dll and have a working module. I like to keep going with it and create a module manifest. We need the manifest if we want to publish to the PSGallery later.

From the root of our project, we can run this command to create the module manifest that we need.

```
PowerShell

$manifestSplat = @{
    Path          = ".\${module}\${module}.psd1"
    Author        = 'Kevin Marquette'
    NestedModules = @('bin\MyModule.dll')
    RootModule     = "\${module}.psm1"
    FunctionsToExport = @('Resolve-MyCmdlet')
}
New-ModuleManifest @manifestSplat
```

I'm also going to create an empty root module for future PowerShell functions.

```
PowerShell

Set-Content -Value '' -Path ".\${module}\${module}.psm1"
```

This allows me to mix both normal PowerShell functions and binary cmdlets in the same project.

Building the full module

I compile everything together into an output folder. We need to create a build script to do that. I would normally add this to an `Invoke-Build` script, but we can keep it simple for this example. Add this to a `build.ps1` at the root of the project.

```
PowerShell

$module = 'MyModule'
Push-Location $PSScriptRoot

dotnet build $PSScriptRoot\src -o $PSScriptRoot\output\$module\bin
```

```
Copy-Item "$PSScriptRoot\$module\*" "$PSScriptRoot\output\$module" -Recurse  
-Force  
  
Import-Module "$PSScriptRoot\Output\$module\$module.psd1"  
Invoke-Pester "$PSScriptRoot\Tests"
```

These commands build our DLL and place it into our `output\$module\bin` folder. It then copies the other module files into place.

```
Output  
└── MyModule  
    ├── MyModule.psd1  
    ├── MyModule.psm1  
    └── bin  
        ├── MyModule.deps.json  
        ├── MyModule.dll  
        └── MyModule.pdb
```

At this point, we can import our module with the `psd1` file.

```
PowerShell  
  
Import-Module ".\Output\$module\$module.psd1"
```

From here, we can drop the `.\Output\$module` folder into our `$Env:PSModulePath` directory and it autoloads our command whenever we need it.

Update: dotnet new PSModule

I learned that the `dotnet` tool has a `PSModule` template.

All the steps that I outlined above are still valid, but this template cuts many of them out. It's still a fairly new template that's still getting some polish placed on it. Expect it to keep getting better from here.

This is how you use install and use the `PSModule` template.

```
PowerShell  
  
dotnet new -i Microsoft.PowerShell.Standard.Module.Template  
dotnet new psmodule  
dotnet build  
Import-Module "bin\Debug\netstandard2.0\$module.dll"  
Get-Module $module
```

This minimally-viable template takes care of adding the .NET SDK, PowerShell Standard Library, and creates an example class in the project. You can build it and run it right away.

Important details

Before we end this article, here are a few other details worth mentioning.

Unloading DLLs

Once a binary module is loaded, you can't really unload it. The DLL file is locked until you unload it. This can be annoying when developing because every time you make a change and want to build it, the file is often locked. The only reliable way to resolve this is to close the PowerShell session that loaded the DLL.

VS Code reload window action

I do most of my PowerShell dev work in [VS Code](#). When I'm working on a binary module (or a module with classes), I've gotten into the habit of reloading VS Code every time I build. `Ctrl + Shift + P` pops the command window and `Reload Window` is always at the top of my list.

Nested PowerShell sessions

One other option is to have good Pester test coverage. Then you can adjust the `build.ps1` script to start a new PowerShell session, perform the build, run the tests, and close the session.

Updating installed modules

This locking can be annoying when trying to update your locally installed module. If any session has it loaded, you have to go hunt it down and close it. This is less of an issue when installing from a PSGallery because module versioning places the new one in a different folder.

You can set up a local PSGallery and publish to that as part of your build. Then do your local install from that PSGallery. This sounds like a lot of work, but this can be as simple as starting a docker container. I cover a way to do that in my post on [Using a NuGet server for a PSRepository](#).

Final thoughts

I didn't touch on the C# syntax for creating a cmdlet, but there is plenty of documentation on it in the [Windows PowerShell SDK](#). It's definitely something worth experimenting with as a stepping stone into more serious C#.

Choosing the right PowerShell NuGet package for your .NET project

Article • 11/17/2022

Alongside the `pwsh` executable packages published with each PowerShell release, the PowerShell team also maintains several packages available on [NuGet](#). These packages allow targeting PowerShell as an API platform in .NET.

As a .NET application that provides APIs and expects to load .NET libraries implementing its own (binary modules), it's essential that PowerShell be available in the form of a NuGet package.

Currently there are several NuGet packages that provide some representation of the PowerShell API surface area. Which package to use with a particular project hasn't always been made clear. This article sheds some light on a few common scenarios for PowerShell-targeting .NET projects and how to choose the right NuGet package to target for your PowerShell-oriented .NET project.

Hosting vs referencing

Some .NET projects seek to write code to be loaded into a pre-existing PowerShell runtime (such as `pwsh`, `powershell.exe`, the PowerShell Integrated Console or the ISE), while others want to run PowerShell in their own applications.

- **Referencing** is for when a project, usually a module, is intended to be loaded into PowerShell. It must be compiled against the APIs that PowerShell provides in order to interact with it, but the PowerShell implementation is supplied by the PowerShell process loading it in. For referencing, a project can use [reference assemblies](#) or the actual runtime assemblies as a compilation target, but must ensure that it does not publish any of these with its build.
- **Hosting** is when a project needs its own implementation of PowerShell, usually because it is a standalone application that needs to run PowerShell. In this case, pure reference assemblies cannot be used. Instead, a concrete PowerShell implementation must be depended upon. Because a concrete PowerShell implementation must be used, a specific version of PowerShell must be chosen for hosting; a single host application cannot multi-target PowerShell versions.

Publishing projects that target PowerShell as a reference

ⓘ Note

We use the term **publish** in this article to refer to running `dotnet publish`, which places a .NET library into a directory with all of its dependencies, ready for deployment to a particular runtime.

In order to prevent publishing project dependencies that are just being used as compilation reference targets, it is recommended to set the `PrivateAssets` attribute:

XML

```
<PackageReference Include="PowerShellStandard.Library" Version="5.1.0.0"
  PrivateAssets="all" />
```

If you forget to do this and use a reference assembly as your target, you may see issues related to using the reference assembly's default implementation instead of the actual implementation. This may take the form of a `NullReferenceException`, since reference assemblies often mock the implementation API by simply returning `null`.

Key kinds of PowerShell-targeting .NET projects

While any .NET library or application can embed PowerShell, there are some common scenarios that use PowerShell APIs:

- **Implementing a PowerShell binary module**

PowerShell binary modules are .NET libraries loaded by PowerShell that must implement PowerShell APIs like the `PSCmdlet` or `CmdletProvider` types in order to expose cmdlets or providers respectively. Because they are loaded in, modules seek to compile against references to PowerShell without publishing it in their build. It's also common for modules to want to support multiple PowerShell versions and platforms, ideally with a minimum of overhead of disk space, complexity, or repeated implementation. See [about_Modules](#) for more information about modules.

- **Implementing a PowerShell Host**

A PowerShell Host provides an interaction layer for the PowerShell runtime. It is a specific form of *hosting*, where a `PSHost` is implemented as a new user interface to PowerShell. For example, the PowerShell ConsoleHost provides a terminal user interface for PowerShell executables, while the PowerShell Editor Services Host and the ISE Host both provide an editor-integrated partially graphical user interface

around PowerShell. While it's possible to load a host onto an existing PowerShell process, it's much more common for a host implementation to act as a standalone PowerShell implementation that redistributes the PowerShell engine.

- **Calling into PowerShell from another .NET application**

As with any application, PowerShell can be called as a subprocess to run workloads. However, as a .NET application, it's also possible to invoke PowerShell in-process to get back full .NET objects for use within the calling application. This is a more general form of *hosting*, where the application holds its own PowerShell implementation for internal use. Examples of this might be a service or daemon running PowerShell to manage machine state or a web application that runs PowerShell on request to do something like manage cloud deployments.

- **Unit testing PowerShell modules from .NET**

While modules and other libraries designed to expose functionality to PowerShell should be primarily tested from PowerShell (we recommend [Pester](#)), sometimes it's necessary to unit test APIs written for a PowerShell module from .NET. This situation involves the module code trying to target a number of PowerShell versions, while testing should run it on specific, concrete implementations.

PowerShell NuGet packages at a glance

In this article, we'll cover the following NuGet packages that expose PowerShell APIs:

- [PowerShellStandard.Library](#), a reference assembly that enables building a single assembly that can be loaded by multiple PowerShell runtimes.
- [Microsoft.PowerShell.SDK](#), the way to target and rehost the whole PowerShell SDK
- The [System.Management.Automation](#) package, the core PowerShell runtime and engine implementation, that can be useful in minimal hosted implementations and for version-specific targeting scenarios.
- The [Windows PowerShell reference assemblies](#), the way to target and effectively rehost Windows PowerShell (PowerShell versions 5.1 and below).

 **Note**

The [PowerShell NuGet](#) package is not a .NET library package at all, but instead provides the PowerShell dotnet global tool implementation. This should not be used by any projects, since it only provides an executable.

PowerShellStandard.Library

The PowerShell Standard library is a reference assembly that captures the intersection of the APIs of PowerShell versions 7, 6 and 5.1. This provides a compile-time-checked API surface to compile .NET code against, allowing .NET projects to target PowerShell versions 7, 6 and 5.1 without risking calling an API that won't be there.

PowerShell Standard is intended for writing PowerShell modules, or other code only intended to be run after loading it into a PowerShell process. Because it is a reference assembly, PowerShell Standard contains no implementation itself, so provides no functionality for standalone applications.

Using PowerShell Standard with different .NET runtimes

PowerShell Standard targets the [.NET Standard 2.0](#) target runtime, which is a façade runtime designed to provide a common surface area shared by .NET Framework and .NET Core. This allows targeting a single runtime to produce a single assembly that will work with multiple PowerShell versions, but has the following consequences:

- The PowerShell loading the module or library must be running a minimum of .NET 4.6.1; .NET 4.6 and .NET 4.5.2 do not support .NET Standard. Note that a newer Windows PowerShell version does not mean a newer .NET Framework version; Windows PowerShell 5.1 may run on .NET 4.5.2.
- In order to work with a PowerShell running .NET Framework 4.7.1 or below, the .NET 4.6.1 [NETStandard.Library](#) implementation is required to provide the netstandard.dll and other shim assemblies in older .NET Framework versions.

PowerShell 6+ provides its own shim assemblies for type forwarding from .NET Framework 4.6.1 (and above) to .NET Core. This means that as long as a module uses only APIs that exist in .NET Core, PowerShell 6+ can load and run it when it has been built for .NET Framework 4.6.1 (the `net461` runtime target).

This means that binary modules using PowerShell Standard to target multiple PowerShell versions with a single published DLL have two options:

1. Publishing an assembly built for the `net461` target runtime. This involves:
 - Publishing the project for the `net461` runtime
 - Also compiling against the `netstandard2.0` runtime (without using its build output) to ensure that all APIs used are also present in .NET Core.
2. Publishing an assembly build for the `netstandard2.0` target runtime. This requires:

- Publishing the project for the `netstandard2.0` runtime
- Taking the `net461` dependencies of `NETStandard.Library` and copying them into the project assembly's publish location so that the assembly is type-forwarded correctly in .NET Framework.

To build PowerShell modules or libraries targeting older .NET Framework versions, it may be preferable to target multiple .NET runtimes. This will publish an assembly for each target runtime, and the correct assembly will need to be loaded at module load time (for example with a small `psm1` as the root module).

Testing PowerShell Standard projects in .NET

When it comes to testing your module in .NET test runners like xUnit, remember that compile-time checks can only go so far. You must test your module against the relevant PowerShell platforms.

To test APIs built against PowerShell Standard in .NET, you should add `Microsoft.PowerShell.SDK` as a testing dependency with .NET Core (with the version set to match the desired PowerShell version), and the appropriate Windows PowerShell reference assemblies with .NET Framework.

For more information on PowerShell Standard and using it to write a binary module that works in multiple PowerShell versions, see [this blog post](#). Also see the [PowerShell Standard repository](#) on GitHub.

Microsoft.PowerShell.SDK

`Microsoft.PowerShell.SDK` is a meta-package that pulls together all of the components of the PowerShell SDK into a single NuGet package. A self-contained .NET application can use `Microsoft.PowerShell.SDK` to run arbitrary PowerShell functionality without depending on any external PowerShell installations or libraries.

ⓘ Note

The PowerShell SDK just refers to all the component packages that make up PowerShell, and which can be used for .NET development with PowerShell.

A given `Microsoft.PowerShell.SDK` version contains the concrete implementation of the same version of the PowerShell application; version 7.0 contains the implementation of PowerShell 7.0 and running commands or scripts with it will largely behave like running them in PowerShell 7.0.

Running PowerShell commands from the SDK is mostly, but not totally, the same as running them from `pwsh`. For example, `Start-Job` currently depends on the `pwsh` executable being available, and so will not work with `Microsoft.PowerShell.SDK` by default.

Targeting `Microsoft.PowerShell.SDK` from a .NET application allows you to integrate with all of PowerShell's implementation assemblies, such as `System.Management.Automation`, `Microsoft.PowerShell.Management`, and other module assemblies.

Publishing an application targeting `Microsoft.PowerShell.SDK` will include all these assemblies, and any dependencies PowerShell requires. It will also include other assets that PowerShell required in its build, such as the module manifests for `Microsoft.PowerShell.*` modules and the `ref` directory required by `Add-Type`.

Given the completeness of `Microsoft.PowerShell.SDK`, it's best suited for:

- Implementation of PowerShell hosts.
- xUnit testing of libraries targeting PowerShell reference assemblies.
- Invoking PowerShell in-process from a .NET application.

`Microsoft.PowerShell.SDK` may also be used as a reference target when a .NET project is intended to be used as a module or otherwise loaded by PowerShell, but depends on APIs only present in a particular version of PowerShell. Note that an assembly published against a specific version of `Microsoft.PowerShell.SDK` will only be safe to load and use in that version of PowerShell. To target multiple PowerShell versions with specific APIs, multiple builds are required, each targeting their own version of `Microsoft.PowerShell.SDK`.

Note

The PowerShell SDK is only available for PowerShell versions 6 and up. To provide equivalent functionality with Windows PowerShell, use the Windows PowerShell reference assemblies described below.

System.Management.Automation

The `System.Management.Automation` package is the heart of the PowerShell SDK. It exists on NuGet, primarily, as an asset for `Microsoft.PowerShell.SDK` to pull in. However, it can also be used directly as a package for smaller hosting scenarios and version-targeting modules.

Specifically, the `System.Management.Automation` package may be a preferable provider of PowerShell functionality when:

- You're only looking to use PowerShell language functionality (in the `System.Management.Automation.Language` namespace) like the PowerShell parser, AST, and AST visitor APIs (for example for static analysis of PowerShell).
- You only wish to execute specific commands from the `Microsoft.PowerShell.Core` module and can execute them in a session state created with the `CreateDefault2` factory method.

Additionally, `System.Management.Automation` is a useful reference assembly when:

- You wish to target APIs that are only present within a specific PowerShell version
- You won't be depending on types occurring outside the `System.Management.Automation` assembly (for example, types exported by cmdlets in `Microsoft.PowerShell.*` modules).

Windows PowerShell reference assemblies

For PowerShell versions 5.1 and older (Windows PowerShell), there is no SDK to provide an implementation of PowerShell, since Windows PowerShell's implementation is a part of Windows.

Instead, the Windows PowerShell reference assemblies provide both reference targets and a way to rehost Windows PowerShell, acting the same as the PowerShell SDK does for versions 6 and up.

Rather than being differentiated by version, Windows PowerShell reference assemblies have a different package for each version of Windows PowerShell:

- [PowerShell 5.1 ↗](#)
- [PowerShell 4 ↗](#)
- [PowerShell 3 ↗](#)

Information on how to use the Windows PowerShell reference assemblies can be found in the [Windows PowerShell SDK](#).

Real-world examples using these NuGet packages

Different PowerShell tooling projects target different PowerShell NuGet packages depending on their needs. Listed here are some notable examples.

PSReadLine

[PSReadLine](#), the PowerShell module that provides much of PowerShell's rich console experience, targets PowerShell Standard as a dependency rather than a specific PowerShell version, and targets the `net461` .NET runtime in its [csproj](#).

PowerShell 6+ supplies its own shim assemblies that allow a DLL targeting the `net461` runtime to "just work" when loaded in (by redirecting binding to .NET Framework's `mscorlib.dll` to the relevant .NET Core assembly).

This simplifies PSReadLine's module layout and delivery significantly, since PowerShell Standard ensures the only APIs used will be present in both PowerShell 5.1 and PowerShell 6+, while also allowing the module to ship with only a single assembly.

The .NET 4.6.1 target does mean that Windows PowerShell running on .NET 4.5.2 and .NET 4.6 is not supported though.

PowerShell Editor Services

[PowerShell Editor Services](#) (PSES) is the backend for the [PowerShell extension](#) for [Visual Studio Code](#), and is actually a form of PowerShell module that gets loaded by a PowerShell executable and then takes over that process to rehost PowerShell within itself while also providing Language Service Protocol and Debug Adapter features.

PSES provides concrete implementation targets for `netcoreapp2.1` to target PowerShell 6+ (since PowerShell 7's `netcoreapp3.1` runtime is backwards compatible) and `net461` to target Windows PowerShell 5.1, but contains most of its logic in a second assembly that targets `netstandard2.0` and PowerShell Standard. This allows it to pull in dependencies required for .NET Core and .NET Framework platforms, while still simplifying most of the codebase behind a uniform abstraction.

Because it is built against PowerShell Standard, PSES requires a runtime implementation of PowerShell in order to be tested correctly. To do this, [PSES's xUnit](#) tests pull in `Microsoft.PowerShell.SDK` and `Microsoft.PowerShell.5.ReferenceAssemblies` in order to provide a PowerShell implementation in the test environment.

As with PSReadLine, PSES cannot support .NET 4.6 and below, but it [performs a check](#) at runtime before calling any of the APIs that could cause a crash on the lower .NET Framework runtimes.

PSScriptAnalyzer

[PSScriptAnalyzer](#), the linter for PowerShell, must target syntactic elements only introduced in certain versions of PowerShell. Because recognition of these syntactic elements is accomplished by implementing an [AstVisitor2](#), it's not possible to use `PowerShellStandard` and also implement AST visitor methods for newer PowerShell syntaxes.

Instead, PSScriptAnalyzer [targets each PowerShell version](#) as a build configuration, and produces a separate DLL for each of them. This increases build size and complexity, but allows:

- Version-specific API targeting
- Version-specific functionality to be implemented with essentially no runtime cost
- Total support for Windows PowerShell all the way down to .NET Framework 4.5.2

Summary

In this article, we've listed and discussed the NuGet packages available to target when implementing a .NET project that uses PowerShell, and the reasons you might have for using one over another.

If you've skipped to the summary, some broad recommendations are:

- PowerShell **modules** should compile against PowerShell Standard if they only require APIs common to different PowerShell versions.
- PowerShell **hosts and applications** that need to run PowerShell internally should target the PowerShell SDK for PowerShell 6+ or the relevant Windows PowerShell reference assemblies for Windows PowerShell.
- PowerShell modules that need **version-specific APIs** should target the PowerShell SDK or Windows PowerShell reference assemblies for the required PowerShell versions, using them as reference assemblies (that is, not publishing the PowerShell dependencies).

Resolving PowerShell module assembly dependency conflicts

Article • 09/08/2022

When writing a binary PowerShell module in C#, it's natural to take dependencies on other packages or libraries to provide functionality. Taking dependencies on other libraries is desirable for code reuse. PowerShell always loads assemblies into the same context. This presents issues when a module's dependencies conflict with already-loaded DLLs and may prevent using two otherwise unrelated modules in the same PowerShell session.

If you've had this problem, you've seen an error message like this:

```
C:\Users\Robert Holt\Documents\Dev\sandbox\ModuleConflict
> ipmo .\FunCsv\FunCsv
C:\Users\Robert Holt\Documents\Dev\sandbox\ModuleConflict
> Get-FunCsv -Path .\ex.csv
C:\Users\Robert Holt\Documents\Dev\sandbox\ModuleConflict
> ipmo .\TotallyUnrelatedModule\TotallyUnrelatedModule\
C:\Users\Robert Holt\Documents\Dev\sandbox\ModuleConflict
> Test-UnrelatedThing -path .\ex.csv
Test-UnrelatedThing: Could not load file or assembly 'CsvHelper, Version=15.0.0.0, Culture=neutral, PublicKeyToken=8c4959082be5c823'. Could not find or load a specific file. (0x80131621)
```

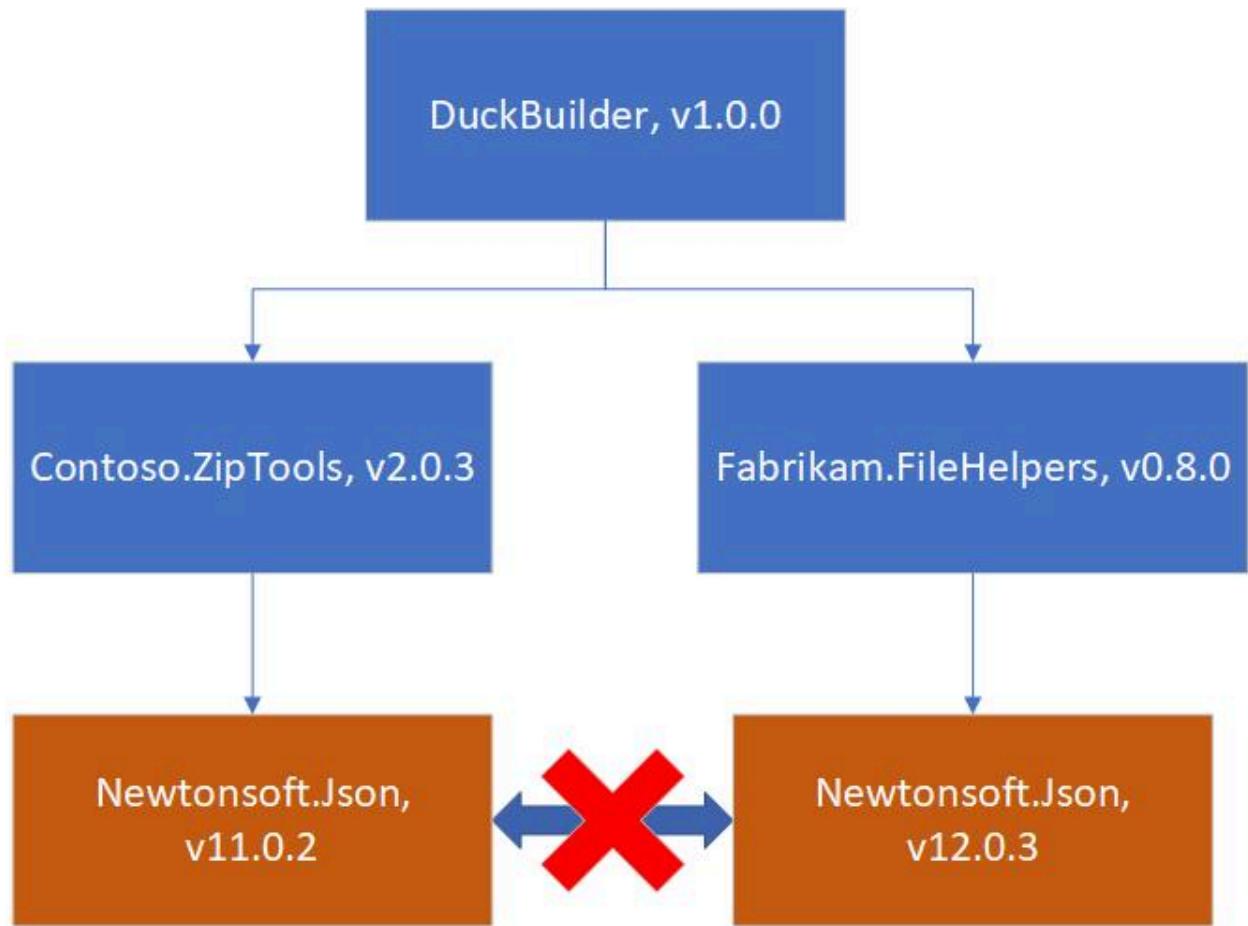
This article looks at some ways dependency conflicts occur in PowerShell and ways to mitigate dependency conflict issues. Even if you're not a module author, there are some tricks in here that might help you with dependency conflicts occurring in modules that you use.

Why do dependency conflicts occur?

In .NET, dependency conflicts occur when two versions of the same assembly are loaded into the same *Assembly Load Context*. This term means slightly different things on different .NET platforms, which is covered [later](#) in this article. This conflict is a common problem that occurs in any software where versioned dependencies are used.

Conflict issues are compounded by the fact that a project almost never deliberately or directly depends on two versions of the same dependency. Instead, the project has two or more dependencies that each require a different version of the same dependency.

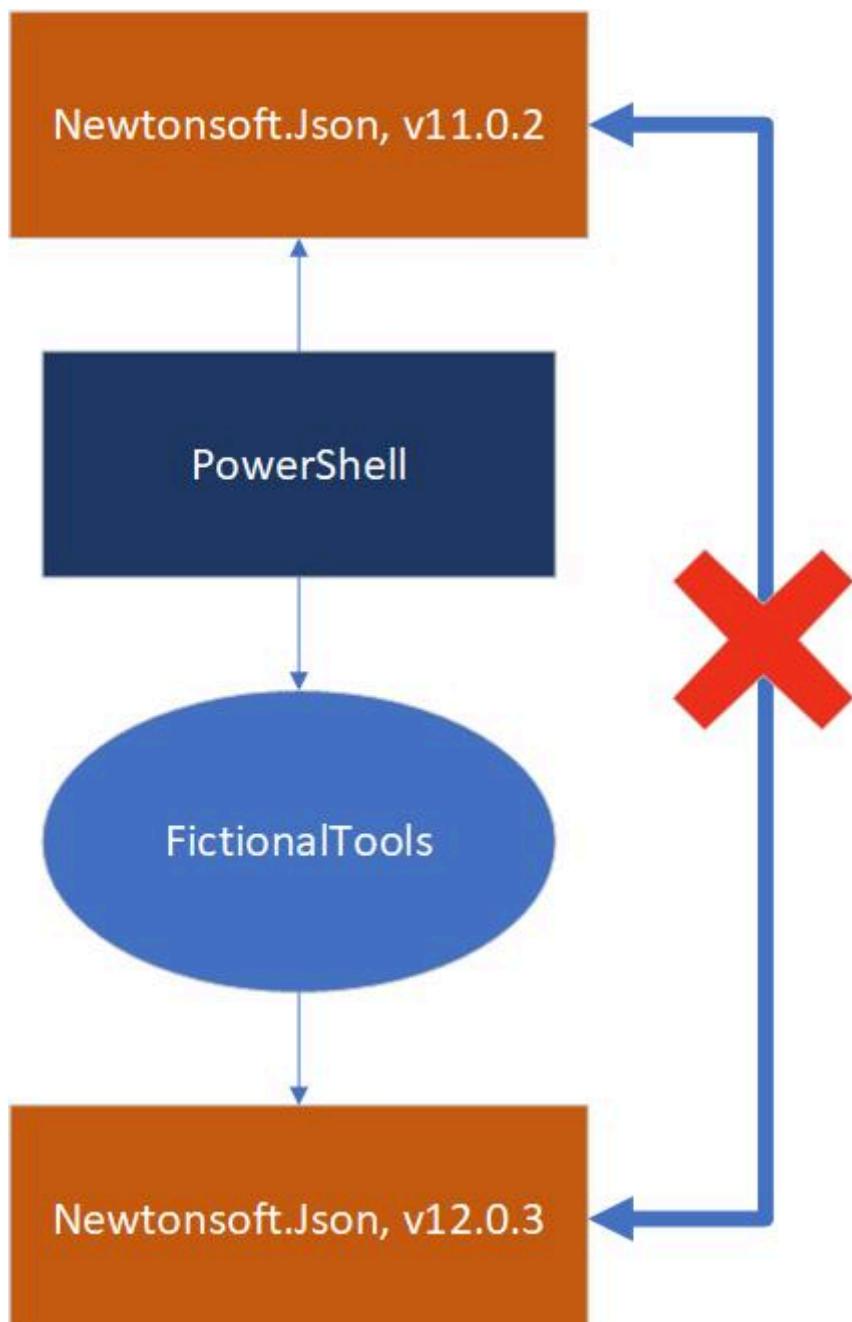
For example, say your .NET application, `DuckBuilder`, brings in two dependencies, to perform parts of its functionality and looks like this:



Because `Contoso.ZipTools` and `Fabrikam.FileHelpers` both depend on different versions of `Newtonsoft.Json`, there may be a dependency conflict depending on how each dependency is loaded.

Conflicting with PowerShell's dependencies

In PowerShell, the dependency conflict issue is magnified because PowerShell's own dependencies are loaded into the same shared context. This means the PowerShell engine and all loaded PowerShell modules must not have conflicting dependencies. A classic example of this is `Newtonsoft.Json`:



In this example, the module `FictionalTools` depends on `Newtonsoft.Json` version `12.0.3`, which is a newer version of `Newtonsoft.Json` than `11.0.2` that ships in the example `PowerShell`.

! Note

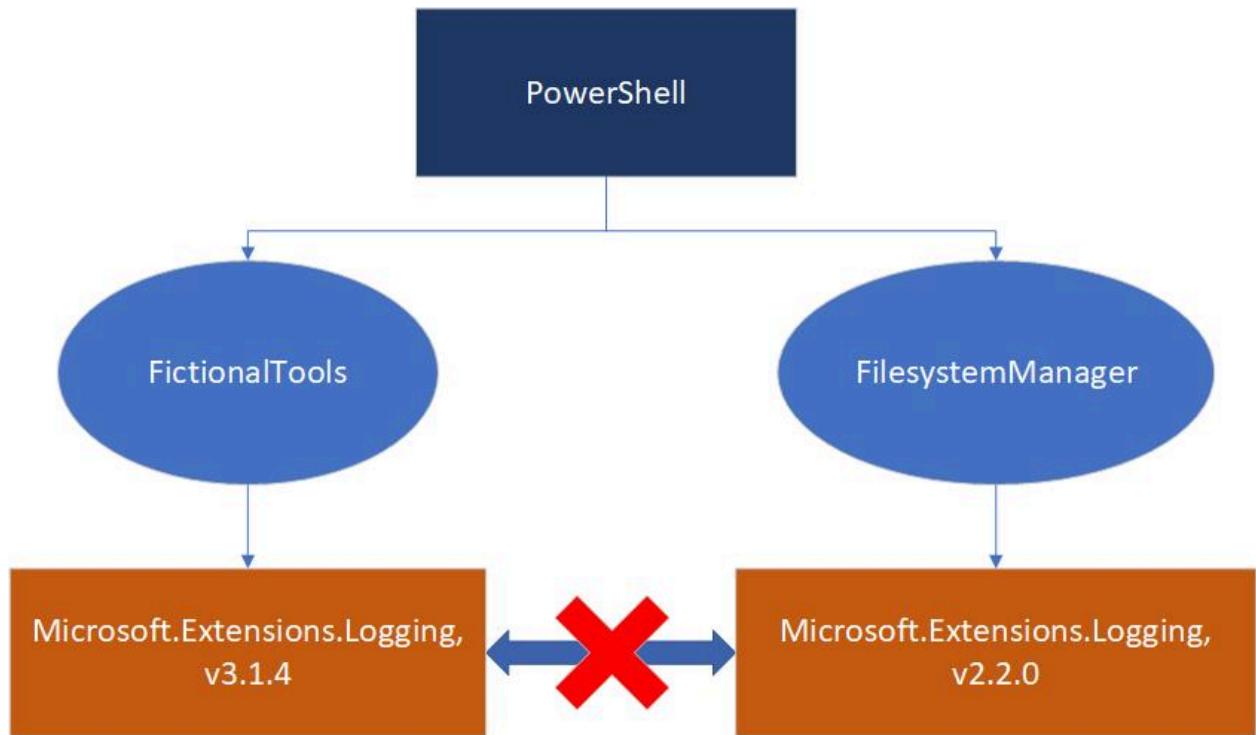
This is an example. PowerShell 7.0 currently ships with `Newtonsoft.Json 12.0.3`. Newer versions of PowerShell have newer versions of `Newtonsoft.Json`.

Because the module depends on a newer version of the assembly, it won't accept the version that PowerShell already has loaded. But because PowerShell has already loaded a version of the assembly, the module can't load its own version using the conventional load mechanism.

Conflicting with another module's dependencies

Another common scenario in PowerShell is that a module is loaded that depends on one version of an assembly, and then another module is loaded later that depends on a different version of that assembly.

This often looks like the following:



In this case, the `FictionalTools` module requires a newer version of `Microsoft.Extensions.Logging` than the `FilesystemManager` module.

Imagine these modules load their dependencies by placing the dependency assemblies in the same directory as the root module assembly. This allows .NET to implicitly load them by name. If we're running PowerShell 7.0 (on top of .NET Core 3.1), we can load and run `FictionalTools`, then load and run `FilesystemManager` without issue. However, in a new session, if we load and run `FilesystemManager`, then load `FictionalTools`, we get a `FileNotFoundException` from the `FictionalTools` command because it requires a newer version of `Microsoft.Extensions.Logging` than the one loaded. `FictionalTools` can't load the version needed because an assembly of the same name has already been loaded.

PowerShell and .NET

PowerShell runs on the .NET platform, which is responsible for resolving and loading assembly dependencies. We must understand how .NET operates here to understand dependency conflicts.

We must also confront the fact that different versions of PowerShell run on different .NET implementations. In general, PowerShell 5.1 and below run on .NET Framework, while PowerShell 6 and above run on .NET Core. These two implementations of .NET load and handle assemblies differently. This means that resolving dependency conflicts can vary depending on the underlying .NET platform.

Assembly Load Contexts

In .NET, an *Assembly Load Context* (ALC) is a runtime namespace into which assemblies are loaded. The assemblies' names must be unique. This concept allows assemblies to be uniquely resolved by name in each ALC.

Assembly reference loading in .NET

The semantics of assembly loading depend on both the .NET implementation (.NET Core vs .NET Framework) and the .NET API used to load a particular assembly. Rather than go into detail here, there are links in the [Further reading](#) section that go into great detail on how .NET assembly loading works in each .NET implementation.

In this article we'll refer to the following mechanisms:

- Implicit assembly loading (effectively `Assembly.Load(AssemblyName)`), when .NET implicitly tries to load an assembly by name from a static assembly reference in .NET code.
- `Assembly.LoadFrom()`, a plugin-oriented loading API that adds handlers to resolve dependencies of the loaded DLL. This method may not resolve dependencies the way we want.
- `Assembly.LoadFile()`, a basic loading API intended to load only the assembly asked for and does not handle any dependencies.

Differences in .NET Framework vs .NET Core

The way these APIs work has changed in subtle ways between .NET Core and .NET Framework, so it's worth reading through the included [links](#). Importantly, Assembly Load Contexts and other assembly resolution mechanisms have changed between .NET Framework and .NET Core.

In particular, .NET Framework has the following features:

- The Global Assembly Cache, for machine-wide assembly resolution
- Application Domains, which work like in-process sandboxes for assembly isolation, but also present a serialization layer to contend with

- A limited assembly load context model that has a fixed set of assembly load contexts, each with their own behavior:
 - The default load context, where assemblies are loaded by default
 - The load-from context, for loading assemblies manually at runtime
 - The reflection-only context, for safely loading assemblies to read their metadata without running them
 - The mysterious void that assemblies loaded with `Assembly.LoadFile(string path)` and `Assembly.Load(byte[] asmBytes)` live in

For more information, see [Best Practices for Assembly Loading](#).

.NET Core (and .NET 5+) has replaced this complexity with a simpler model:

- No Global Assembly Cache. Applications bring all their own dependencies. This removes an external factor for dependency resolution in applications, making dependency resolution more reproducible. PowerShell, as the plugin host, complicates this slightly for modules. Its dependencies in `$PSHOME` are shared with all modules.
- Only one Application Domain, and no ability to create new ones. The Application Domain concept is maintained in .NET to be the global state of the .NET process.
- A new, extensible Assembly Load Context (ALC) model. Assembly resolution can be namespaced by putting it in a new ALC. .NET processes begin with a single default ALC into which all assemblies are loaded (except for those loaded with `Assembly.LoadFile(string)` and `Assembly.Load(byte[])`). But the process can create and define its own custom ALCs with its own loading logic. When an assembly is loaded, the first ALC it's loaded into is responsible for resolving its dependencies. This creates opportunities to implement powerful .NET plugin loading mechanisms.

In both implementations, assemblies are loaded lazily. This means that they're loaded when a method requiring their type is run for the first time.

For example, here are two versions of the same code that load a dependency at different times.

The first always loads its dependency when `Program.GetRange()` is called, because the dependency reference is lexically present within the method:

C#

```
using Dependency.Library;

public static class Program
{
```

```

public static List<int> GetRange(int limit)
{
    var list = new List<int>();
    for (int i = 0; i < limit; i++)
    {
        if (i >= 20)
        {
            // Dependency.Library will be loaded when GetRange is run
            // because the dependency call occurs directly within the
method
            DependencyApi.Use();
        }

        list.Add(i);
    }
    return list;
}

```

The second loads its dependency only if the `limit` parameter is 20 or more, because of the internal indirection through a method:

```

C#

using Dependency.Library;

public static class Program
{
    public static List<int> GetNumbers(int limit)
    {
        var list = new List<int>();
        for (int i = 0; i < limit; i++)
        {
            if (i >= 20)
            {
                // Dependency.Library is only referenced within
                // the UseDependencyApi() method,
                // so will only be loaded when limit >= 20
                UseDependencyApi();
            }

            list.Add(i);
        }
        return list;
    }

    private static void UseDependencyApi()
    {
        // Once UseDependencyApi() is called, Dependency.Library is loaded
        DependencyApi.Use();
    }
}

```

This is a good practice since it minimizes the memory and filesystem I/O and uses the resources more efficiently. The unfortunate side effect of this is that we won't know that the assembly fails to load until we reach the code path that tries to load the assembly.

It can also create a timing condition for assembly load conflicts. If two parts of the same program try to load different versions of the same assembly, the version loaded depends on which code path is run first.

For PowerShell, this means that the following factors can affect an assembly load conflict:

- Which module was loaded first?
- Was the code path that uses the dependency library run?
- Does PowerShell load a conflicting dependency at startup or only under certain code paths?

Quick fixes and their limitations

In some cases, it's possible to make small adjustments to your module and fix things with minimal effort. But these solutions tend to come with caveats. While they may apply to your module, they won't work for every module.

Change your dependency version

The simplest way to avoid dependency conflicts is to agree on a dependency. This may be possible when:

- Your conflict is with a direct dependency of your module and you control the version.
- Your conflict is with an indirect dependency, but you can configure your direct dependencies to use a workable indirect dependency version.
- You know the conflicting version and can rely on it not changing.

The **Newtonsoft.Json** package is a good example of this last scenario. This is a dependency of PowerShell 6 and above, and isn't used in Windows PowerShell. Meaning a simple way to resolve versioning conflicts is to target the lowest version of **Newtonsoft.Json** across the PowerShell versions you wish to target.

For example, PowerShell 6.2.6 and PowerShell 7.0.2 both currently use **Newtonsoft.Json** version 12.0.3. To create a module targeting Windows PowerShell, PowerShell 6, and PowerShell 7, you would target **Newtonsoft.Json 12.0.3** as a dependency and include it

in your built module. When the module is loaded in PowerShell 6 or 7, PowerShell's own **Newtonsoft.Json** assembly is already loaded. Since it's the version required for your module, resolution succeeds. In Windows PowerShell, the assembly isn't already present in PowerShell, so it's loaded from your module folder instead.

Generally, when targeting a concrete PowerShell package, like **Microsoft.PowerShell.Sdk** or **System.Management.Automation**, NuGet should be able to resolve the right dependency versions required. Targeting both Windows PowerShell and PowerShell 6+ becomes more difficult because you must choose between targeting multiple frameworks or **PowerShellStandard.Library**.

Circumstances where pinning to a common dependency version won't work include:

- The conflict is with an indirect dependency, and none of your dependencies can be configured to use a common version.
- The other dependency version is likely to change often, so settling on a common version is only a short-term fix.

Use the dependency out of process

This solution is more for module users than module authors. This is a solution to use when confronted with a module that won't work due to an existing dependency conflict.

Dependency conflicts occur because two versions of the same assembly are loaded into the same .NET process. A simple solution is to load them into different processes, as long as you can still use the functionality from both together.

In PowerShell, there are several ways to achieve this:

- Invoke PowerShell as a subprocess

To run a PowerShell command out of the current process, start a new PowerShell process directly with the command call:

```
PowerShell  
pwsh -c 'Invoke-ConflictingCommand'
```

The main limitation here is that restructuring the result can be trickier or more error prone than other options.

- The PowerShell job system

The PowerShell job system also runs commands out of process, by sending commands to a new PowerShell process and returning the results:

```
PowerShell  
  
$result = Start-Job { Invoke-ConflictingCommand } | Receive-Job -Wait
```

In this case, you just need to be sure that any variables and state are passed in correctly.

The job system can also be slightly cumbersome when running small commands.

- PowerShell remoting

When it's available, PowerShell remoting can be a useful way to run commands out of process. With remoting, you can create a fresh **PSSession** in a new process, call its commands over PowerShell remoting, then use the results locally with the other modules containing the conflicting dependencies.

An example might look like this:

```
PowerShell  
  
# Create a local PowerShell session  
# where the module with conflicting assemblies will be loaded  
$s = New-PSSession  
  
# Import the module with the conflicting dependency via remoting,  
# exposing the commands locally  
Import-Module -PSSession $s -Name ConflictingModule  
  
# Run a command from the module with the conflicting dependencies  
Invoke-ConflictingCommand
```

- Implicit remoting to Windows PowerShell

Another option in PowerShell 7 is to use the `-UseWindowsPowerShell` flag on `Import-Module`. This imports the module through a local remoting session into Windows PowerShell:

```
PowerShell  
  
Import-Module -Name ConflictingModule -UseWindowsPowerShell
```

Be aware that modules may not be compatible with or may work differently with Windows PowerShell.

When out-of-process invocation should not be used

As a module author, out-of-process command invocation is difficult to bake into a module and may have edge cases that cause issues. In particular, remoting and jobs may not be available in all environments where your module needs to work. However, the general principle of moving the implementation out of process and allowing the PowerShell module to be a thinner client, may still be applicable.

As a module user, there are cases where out-of-process invocation won't work:

- When PowerShell remoting is unavailable because you don't have privileges to use it or it is not enabled.
- When a particular .NET type is needed from output as input to a method or another command. Commands running over PowerShell remoting emit serialized objects rather than strongly-typed .NET objects. This means that method calls and strongly typed APIs don't work with the output of commands imported over remoting.

More robust solutions

The previous solutions all had scenarios and modules that don't work. However, they also have the virtue of being relatively simple to implement correctly. The following solutions are more robust, but require more effort to implement correctly and can introduce subtle bugs if not written carefully.

Loading through .NET Core Assembly Load Contexts

[Assembly Load Contexts](#) (ALCs) were introduced in .NET Core 1.0 to specifically address the need to load multiple versions of the same assembly into the same runtime.

Within .NET, they offer the most robust solution to the problem of loading conflicting versions of an assembly. However, custom ALCs are not available in .NET Framework. This means that this solution only works in PowerShell 6 and above.

Currently, the best example of using an ALC for dependency isolation in PowerShell is in PowerShell Editor Services, the language server for the PowerShell extension for Visual Studio Code. An [ALC is used](#) to prevent PowerShell Editor Services' own dependencies from clashing with those in PowerShell modules.

Implementing module dependency isolation with an ALC is conceptually difficult, but we will work through a minimal example. Imagine we have a simple module that is only intended to work in PowerShell 7. The source code is organized as follows:

```
+ AlcModule.psd1
+ src/
  + TestAlcModuleCommand.cs
  + AlcModule.csproj
```

The cmdlet implementation looks like this:

C#

```
using Shared.Dependency;

namespace AlcModule
{
    [Cmdlet(VerbsDiagnostic.Test, "AlcModule")]
    public class TestAlcModuleCommand : Cmdlet
    {
        protected override void EndProcessing()
        {
            // Here's where our dependency gets used
            Dependency.Use();
            // Something trivial to make our cmdlet do *something*
            WriteObject("done!");
        }
    }
}
```

The (heavily simplified) manifest, looks like this:

PowerShell

```
@{
    Author = 'Me'
    ModuleVersion = '0.0.1'
    RootModule = 'AlcModule.dll'
    CmdletsToExport = @('Test-AlcModule')
    PowerShellVersion = '7.0'
}
```

And the `csproj` looks like this:

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Shared.Dependency" Version="1.0.0" />
```

```
<PackageReference Include="Microsoft.PowerShell.Sdk" Version="7.0.1"
PrivateAssets="all" />
</ItemGroup>
</Project>
```

When we build this module, the generated output has the following layout:

```
AlcModule/
+ AlcModule.psd1
+ AlcModule.dll
+ Shared.Dependency.dll
```

In this example, our problem is in the `Shared.Dependency.dll` assembly, which is our imaginary conflicting dependency. This is the dependency that we need to put behind an ALC so that we can use the module-specific version.

We need to re-engineer the module so that:

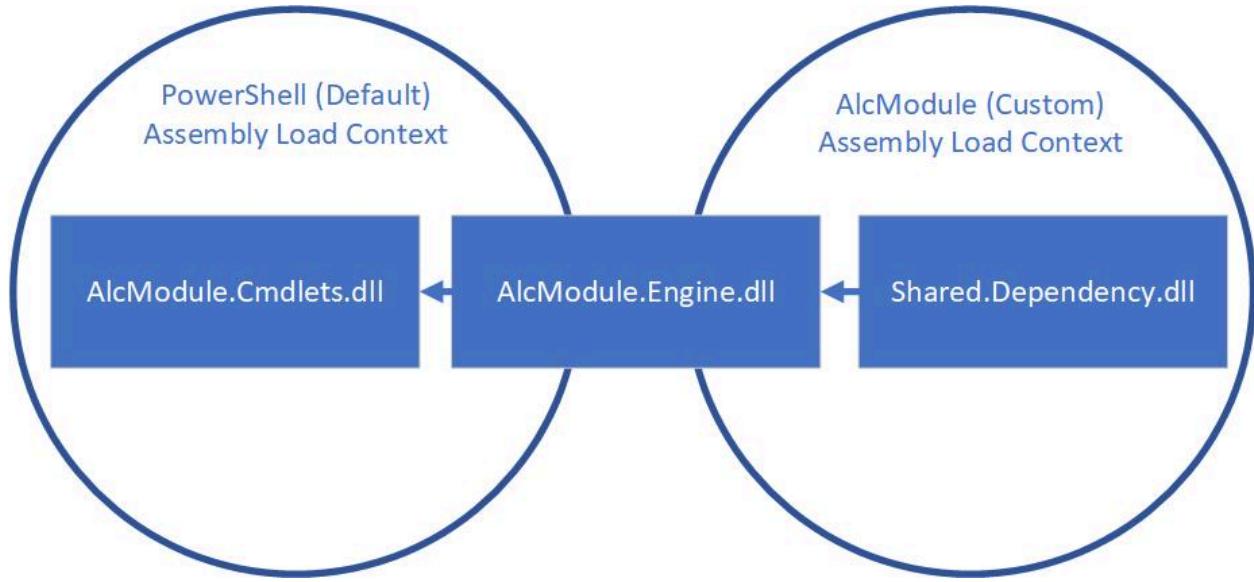
- Module dependencies are only loaded into our custom ALC, and not into PowerShell's ALC, so there can be no conflict. Moreover, as we add more dependencies to our project, we don't want to continuously add more code to keep loading working. Instead, we want reusable, generic dependency resolution logic.
- Loading the module still works as normal in PowerShell. Cmdlets and other types that the PowerShell module system needs are defined within PowerShell's own ALC.

To mediate these two requirements, we must break up our module into two assemblies:

- A cmdlets assembly, `AlcModule.Cmdlets.dll`, that contains definitions of all the types that PowerShell's module system needs to load our module correctly. Namely, any implementations of the `Cmdlet` base class and the class that implements `IModuleAssemblyInitializer`, which sets up the event handler for `AssemblyLoadContext.Default.Resolving` to properly load `AlcModule.Engine.dll` through our custom ALC. Since PowerShell 7 deliberately hides types defined in assemblies loaded in other ALCs, any types that are meant to be publicly exposed to PowerShell must also be defined here. Finally, our custom ALC definition needs to be defined in this assembly. Beyond that, as little code as possible should live in this assembly.
- An engine assembly, `AlcModule.Engine.dll`, that handles the actual implementation of the module. Types from this are available in the PowerShell ALC,

but it's initially loaded through our custom ALC. Its dependencies are only loaded into the custom ALC. Effectively, this becomes a *bridge* between the two ALCs.

Using this bridge concept, our new assembly situation looks like this:



To make sure the default ALC's dependency probing logic doesn't resolve the dependencies to be loaded into the custom ALC, we need to separate these two parts of the module in different directories. The new module layout has the following structure:

```
AlcModule/
  AlcModule.Cmdlets.dll
  AlcModule.psd1
  Dependencies/
    + AlcModule.Engine.dll
    + Shared.Dependency.dll
```

To see how the implementation changes, we'll start with the implementation of `AlcModule.Engine.dll`:

```
C#  
  
using Shared.Dependency;  
  
namespace AlcModule.Engine  
{  
    public class AlcEngine  
    {  
        public static void Use()  
        {  
            Dependency.Use();  
        }  
    }  
}
```

```
    }  
}
```

This is a simple container for the dependency, `Shared.Dependency.dll`, but you should think of it as the .NET API for your functionality that the cmdlets in the other assembly wrap for PowerShell.

The cmdlet in `AlcModule.Cmdlets.dll` looks like this:

```
C#  
  
// Reference our module's Engine implementation here  
using AlcModule.Engine;  
  
namespace AlcModule.Cmdlets  
{  
    [Cmdlet(VerbsDiagnostic.Test, "AlcModule")]  
    public class TestAlcModuleCommand : Cmdlet  
    {  
        protected override void EndProcessing()  
        {  
            AlcEngine.Use();  
            WriteObject("done!");  
        }  
    }  
}
```

At this point, if we were to load `AlcModule` and run `Test-AlcModule`, we get a `FileNotFoundException` when the default ALC tries to load `Alc.Engine.dll` to run `EndProcessing()`. This is good, since it means the default ALC can't find the dependencies we want to hide.

Now we need to add code to `AlcModule.Cmdlets.dll` so that it knows how to resolve `AlcModule.Engine.dll`. First we must define our custom ALC to resolve assemblies from our module's `Dependencies` directory:

```
C#  
  
namespace AlcModule.Cmdlets  
{  
    internal class AlcModuleAssemblyLoadContext : AssemblyLoadContext  
    {  
        private readonly string _dependencyDirPath;  
  
        public AlcModuleAssemblyLoadContext(string dependencyDirPath)  
        {  
            _dependencyDirPath = dependencyDirPath;  
        }  
    }  
}
```

```

protected override Assembly Load(AssemblyName assemblyName)
{
    // We do the simple logic here of looking for an assembly of the
    // given name
    // in the configured dependency directory.
    string assemblyPath = Path.Combine(
        _dependencyDirPath,
        $"{assemblyName.Name}.dll");

    if (File.Exists(assemblyPath))
    {
        // The ALC must use inherited methods to load assemblies.
        // Assembly.Load*() won't work here.
        return LoadFromAssemblyPath(assemblyPath);
    }

    // For other assemblies, return null to allow other resolutions
    // to continue.
    return null;
}
}
}

```

Then we need to hook up our custom ALC to the default ALC's `Resolving` event, which is the ALC version of the `AssemblyResolve` event on Application Domains. This event is fired to find `AlcModule.Engine.dll` when `EndProcessing()` is called.

C#

```

namespace AlcModule.Cmdlets
{
    public class AlcModuleResolveEventHandler : IModuleAssemblyInitializer,
    IModuleAssemblyCleanup
    {
        // Get the path of the dependency directory.
        // In this case we find it relative to the AlcModule.Cmdlets.dll
        location
            private static readonly string s_dependencyDirPath =
        Path.GetFullPath(
            Path.Combine(
                Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location),
                "Dependencies"));

        private static readonly AlcModuleAssemblyLoadContext s_dependencyAlc
        =
            new AlcModuleAssemblyLoadContext(s_dependencyDirPath);

        public void OnImport()
        {
            // Add the Resolving event handler here
        }
    }
}

```

```

        AssemblyLoadContext.Default.Resolving += ResolveAlcEngine;
    }

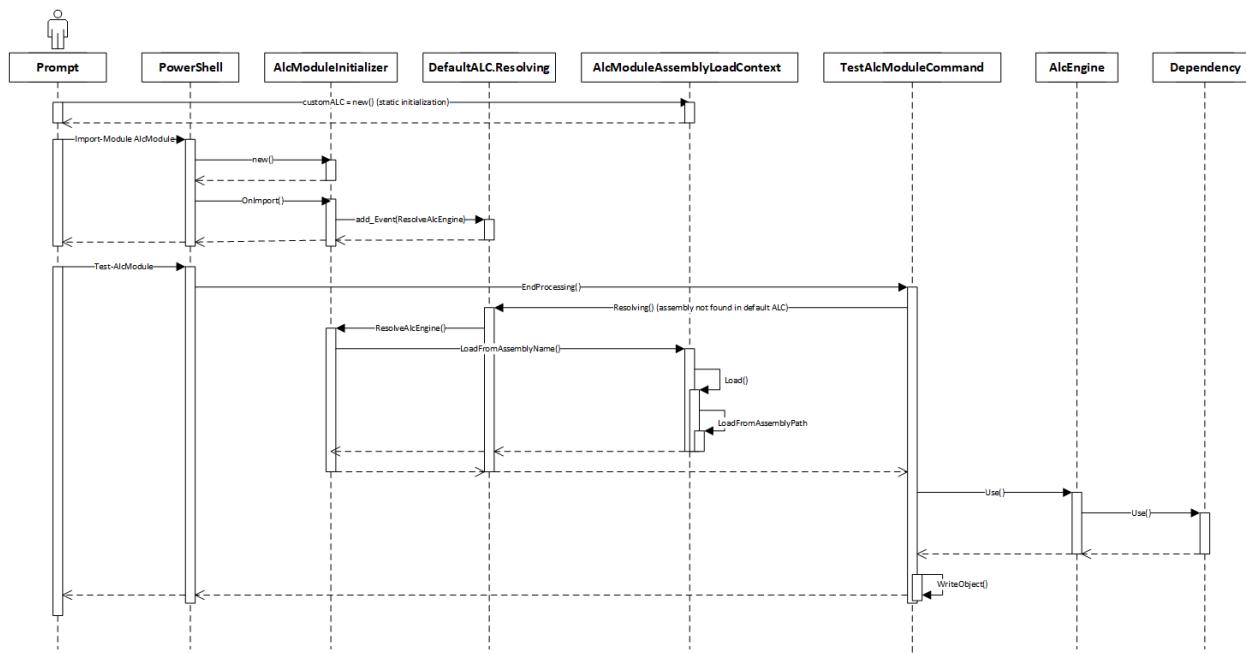
    public void OnRemove(PSModuleInfo psModuleInfo)
    {
        // Remove the Resolving event handler here
        AssemblyLoadContext.Default.Resolving -= ResolveAlcEngine;
    }

    private static Assembly ResolveAlcEngine(AssemblyLoadContext
defaultAlc, AssemblyName assemblyToResolve)
    {
        // We only want to resolve the Alc.Engine.dll assembly here.
        // Because this will be loaded into the custom ALC,
        // all of *its* dependencies will be resolved
        // by the logic we defined for that ALC's implementation.
        //
        // Note that we are safe in our assumption that the name is
enough
        // to distinguish our assembly here,
        // since it's unique to our module.
        // There should be no other AlcModule.Engine.dll on the system.
        if (!assemblyToResolve.Name.Equals("AlcModule.Engine"))
        {
            return null;
        }

        // Allow our ALC to handle the directory discovery concept
        //
        // This is where Alc.Engine.dll is loaded into our custom ALC
        // and then passed through into PowerShell's ALC,
        // becoming the bridge between both
        return s_dependencyAlc.LoadFromAssemblyName(assemblyToResolve);
    }
}

```

With the new implementation, take a look at the sequence of calls that occurs when the module is loaded and `Test-AlcModule` is run:



Some points of interest are:

- The `IModuleAssemblyInitializer` is run first when the module loads and sets the `Resolving` event.
- We don't load the dependencies until `Test-AlcModule` is run and its `EndProcessing()` method is called.
- When `EndProcessing()` is called, the default ALC fails to find `AlcModule.Engine.dll` and fires the `Resolving` event.
- Our event handler hooks up the custom ALC to the default ALC and loads `AlcModule.Engine.dll` only.
- When `AlcEngine.Use()` is called within `AlcModule.Engine.dll`, the custom ALC again kicks in to resolve `Shared.Dependency.dll`. Specifically, it always loads *our* `Shared.Dependency.dll` since it never conflicts with anything in the default ALC and only looks in our `Dependencies` directory.

Assembling the implementation, our new source code layout looks like this:

```
+ AlcModule.psd1
+ src/
  + AlcModule.Cmdlets/
    | + AlcModule.Cmdlets.csproj
    | + TestAlcModuleCommand.cs
    | + AlcModuleAssemblyLoadContext.cs
    | + AlcModuleInitializer.cs
  |
  + AlcModule.Engine/
```

```
| + AlcModule.Engine.csproj  
| + AlcEngine.cs
```

AlcModule.Cmdlets.csproj looks like:

XML

```
<Project Sdk="Microsoft.NET.Sdk">  
  <PropertyGroup>  
    <TargetFramework>netcoreapp3.1</TargetFramework>  
  </PropertyGroup>  
  <ItemGroup>  
    <ProjectReference Include=".\\AlcModule.Engine\\AlcModule.Engine.csproj" />  
    <PackageReference Include="Microsoft.PowerShell.Sdk" Version="7.0.1" PrivateAssets="all" />  
  </ItemGroup>  
</Project>
```

AlcModule.Engine.csproj looks like this:

XML

```
<Project Sdk="Microsoft.NET.Sdk">  
  <PropertyGroup>  
    <TargetFramework>netcoreapp3.1</TargetFramework>  
  </PropertyGroup>  
  <ItemGroup>  
    <PackageReference Include="Shared.Dependency" Version="1.0.0" />  
  </ItemGroup>  
</Project>
```

So, when we build the module, our strategy is:

- Build `AlcModule.Engine`
- Build `AlcModule.Cmdlets`
- Copy everything from `AlcModule.Engine` into the `Dependencies` directory, and remember what we copied
- Copy everything from `AlcModule.Cmdlets` that wasn't in `AlcModule.Engine` into the base module directory

Since the module layout here is so crucial to dependency separation, here's a build script to use from the source root:

PowerShell

```

param(
    # The .NET build configuration
    [ValidateSet('Debug', 'Release')]
    [string]
    $Configuration = 'Debug'
)

# Convenient reusable constants
$mod = "AlcModule"
$netcore = "netcoreapp3.1"
$copyExtensions = @('.dll', '.pdb')

# Source code locations
$src = "$PSScriptRoot/src"
$engineSrc = "$src/$mod.Engine"
$cmdletsSrc = "$src/$mod.Cmdlets"

# Generated output locations
$outDir = "$PSScriptRoot/out/$mod"
$outDeps = "$outDir/Dependencies"

# Build AlcModule.Engine
Push-Location $engineSrc
dotnet publish -c $Configuration
Pop-Location

# Build AlcModule.Cmdlets
Push-Location $cmdletsSrc
dotnet publish -c $Configuration
Pop-Location

# Ensure out directory exists and is clean
Remove-Item -Path $outDir -Recurse -ErrorAction Ignore
New-Item -Path $outDir -ItemType Directory
New-Item -Path $outDeps -ItemType Directory

# Copy manifest
Copy-Item -Path "$PSScriptRoot/$mod.psd1"

# Copy each Engine asset and remember it
$deps = [System.Collections.Generic.Hashtable[string]]::new()
Get-ChildItem -Path "$engineSrc/bin/$Configuration/$netcore/publish/" |
    Where-Object { $_.Extension -in $copyExtensions } |
        ForEach-Object { [void]$deps.Add($_.Name); Copy-Item -Path $_.FullName -
Destination $outDeps }

# Now copy each Cmdlets asset, not taking any found in Engine
Get-ChildItem -Path "$cmdletsSrc/bin/$Configuration/$netcore/publish/" |
    Where-Object { -not $deps.Contains($_.Name) -and $_.Extension -in
$copyExtensions } |
        ForEach-Object { Copy-Item -Path $_.FullName -Destination $outDir }

```

Finally, we have a general way to isolate our module's dependencies in an Assembly Load Context that remains robust over time as more dependencies are added.

For a more detailed example, go to this [GitHub repository](#). This example demonstrates how to migrate a module to use an ALC, while keeping that module working in .NET Framework. It also shows how to use .NET Standard and PowerShell Standard to simplify the core implementation.

This solution is also used by the [Bicep PowerShell module](#), and the blog post [Resolving PowerShell Module Conflicts](#) is another good read about this solution.

Assembly resolving handler for side-by-side loading

Although being robust, the solution described above requires the module assembly to not directly reference the dependency assemblies, but instead, reference a wrapper assembly that references the dependency assemblies. The wrapper assembly acts like a bridge, forwarding the calls from the module assembly to the dependency assemblies. This makes it usually a non-trivial amount of work to adopt this solution:

- For a new module, this would add additional complexity to the design and implementation
- For an existing module, this would require significant refactoring

There is a simplified solution to achieve side-by-side assembly loading, by hooking up a `Resolving` event with a custom `AssemblyLoadContext` instance. Using this method is easier for the module author but has two limitations. Check out the [PowerShell-ALC-Samples](#) repository for sample code and documentation that describes these limitations and detailed scenarios for this solution.

Important

Do not use `Assembly.LoadFile` for the dependency isolation purpose. Using `Assembly.LoadFile` creates a *Type Identity* issue when another module loads a different version of the same assembly into the default `AssemblyLoadContext`. While this API loads an assembly to a separate `AssemblyLoadContext` instance, the assemblies loaded are discoverable by PowerShell's [type resolution code](#). Therefore, there could be duplicate types with the same fully qualified type name available from two different ALCs.

Custom Application Domains

The final and most extreme option for assembly isolation is to use custom **Application Domains**. **Application Domains** are only available in .NET Framework. They are used to provide in-process isolation between parts of a .NET application. One of the uses is to isolate assembly loads from each other within the same process.

However, **Application Domains** are serialization boundaries. Objects in one application domain can't be referenced and used directly by objects in another application domain. You can work around this by implementing `MarshalByRefObject`. But when you don't control the types, as is often the case with dependencies, it's not possible to force an implementation here. The only solution is to make large architectural changes. The serialization boundary also has serious performance implications.

Because **Application Domains** have this serious limitation, are complicated to implement, and only work in .NET Framework, we won't give an example of how you might use them here. While they're worth mentioning as a possibility, they're not recommended.

If you're interested in trying to use a custom application domain, the following links might help:

- [Conceptual documentation on Application Domains](#)
- [Examples for using Application Domains](#)

Solutions for dependency conflicts that don't work for PowerShell

Finally, we'll address some possibilities that come up when researching .NET dependency conflicts in .NET that can look promising, but generally won't work for PowerShell.

These solutions have the common theme that they are changes to deployment configurations for an environment where you control the application and possibly the entire machine. These solutions are oriented toward scenarios like web servers and other applications deployed to server environments, where the environment is intended to host the application and is free to be configured by the deploying user. They also tend to be very much .NET Framework oriented, meaning they don't work with PowerShell 6 or higher.

If you know that your module is only used in Windows PowerShell 5.1 environments that you have total control over, some of these may be options. In general however, **modules shouldn't modify global machine state like this**. It can break configurations

that cause problems in `powershell.exe`, other modules, or other dependent applications that cause your module to fail in unexpected ways.

Static binding redirect with app.config to force using the same dependency version

.NET Framework applications can take advantage of an `app.config` file to configure some application behaviors declaratively. It's possible to write an `app.config` entry that configures assembly binding to redirect assembly loading to a particular version.

Two issues with this for PowerShell are:

- .NET Core doesn't support `app.config`, so this solution only applies to `powershell.exe`.
- `powershell.exe` is a shared application that lives in the `System32` directory. It's likely that your module won't be able to modify its contents on many systems. Even if it can, modifying the `app.config` could break an existing configuration or affect the loading of other modules.

Setting `codebase` with app.config

For the same reasons, trying to configure the `codebase` setting in `app.config` is not going to work in PowerShell modules.

Installing dependencies to the Global Assembly Cache (GAC)

Another way to resolve dependency version conflicts in .NET Framework is to install dependencies to the GAC, so that different versions can be loaded side-by-side from the GAC.

Again, for PowerShell modules, the chief issues here are:

- The GAC only applies to .NET Framework, so this does not help in PowerShell 6 and above.
- Installing assemblies to the GAC is a modification of global machine state and may cause side-effects in other applications or to other modules. It may also be difficult to do correctly, even when your module has the required access privileges. Getting it wrong could cause serious, machine-wide issues in other .NET applications.

Further reading

There's plenty more to read on .NET assembly version dependency conflicts. Here are some nice jumping off points:

- [.NET: Assemblies in .NET](#)
- [.NET Core: The managed assembly loading algorithm](#)
- [.NET Core: Understanding System.Runtime.Loader.AssemblyLoadContext](#)
- [.NET Core: Discussion about side-by-side assembly loading solutions ↗](#)
- [.NET Framework: Redirecting assembly versions](#)
- [.NET Framework: Best practices for assembly loading](#)
- [.NET Framework: How the runtime locates assemblies](#)
- [.NET Framework: Resolve assembly loads](#)
- [Stack Overflow: Assembly binding redirect, how and why? ↗](#)
- [PowerShell: Discussion about implementing AssemblyLoadContexts ↗](#)
- [PowerShell: Assembly.LoadFile\(\) doesn't load into default AssemblyLoadContext ↗](#)
- [Rick Strahl: When does a .NET assembly dependency get loaded? ↗](#)
- [Jon Skeet: Summary of versioning in .NET ↗](#)
- [Nate McMaster: Deep dive into .NET Core primitives ↗](#)

How to create a command-line predictor

Article • 07/10/2023

PSReadLine 2.1.0 introduced the concept of a smart command-line predictor by implementing the Predictive IntelliSense feature. PSReadLine 2.2.2 expanded on that feature by adding a plugin model that allows you create your own command-line predictors.

Predictive IntelliSense enhances tab completion by providing suggestions, on the command line, as you type. The prediction suggestion appears as colored text following your cursor. This enables you to discover, edit, and execute full commands based on matching predictions from your command history or additional domain-specific plugins.

System requirements

To create and use a plugin predictor, you must be using the following versions of software:

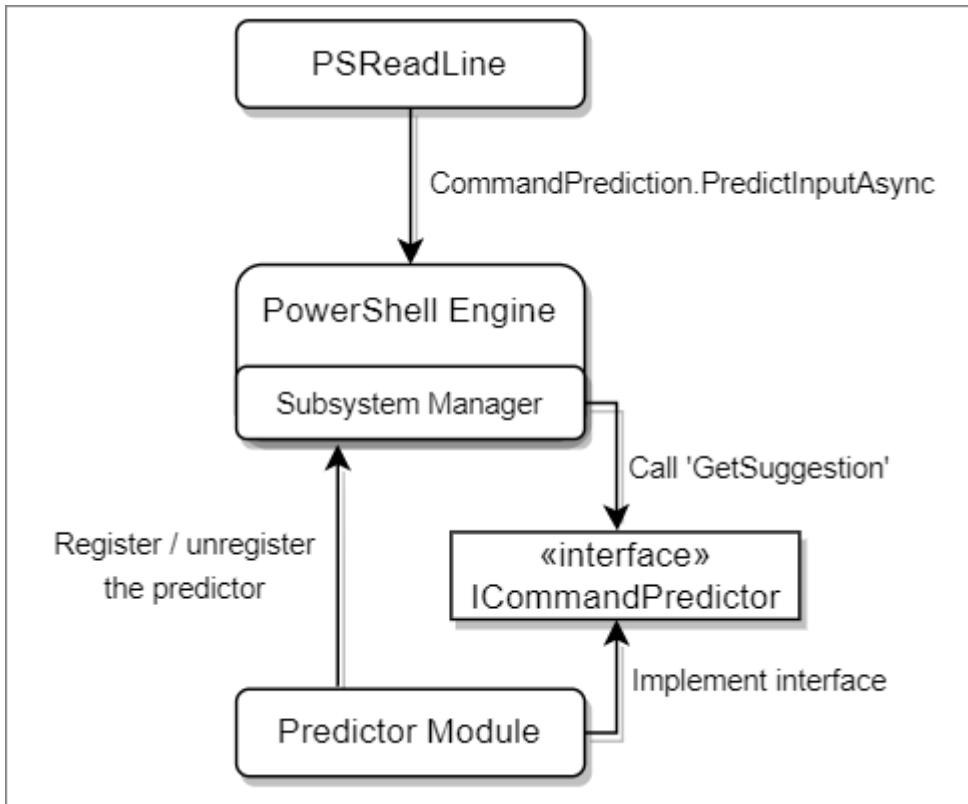
- PowerShell 7.2 (or higher) - provides the APIs necessary for creating a command predictor
- PSReadLine 2.2.2 (or higher) - allows you to configure PSReadLine to use the plugin

Overview of a predictor

A predictor is a PowerShell binary module. The module must implement the `System.Management.Automation.Subsystem.Prediction.ICommandPredictor` interface. This interface declares the methods used to query for prediction results and provide feedback.

A predictor module must register a `CommandPredictor` subsystem with PowerShell's `SubsystemManager` when loaded and unregister itself when unloaded.

The following diagram shows the architecture of a predictor in PowerShell.



Creating the code

To create a predictor, you must have the .NET 6 SDK installed for your platform. For more information on the SDK, see [Download .NET 6.0](#).

Create a new PowerShell module project by following these steps:

1. Use the `dotnet` command-line tool to create a starter classlib project.

```

PowerShell
dotnet new classlib --name SamplePredictor

```

2. Edit the `SamplePredictor.csproj` to contain the following information:

```

XML
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.PowerShell.SDK"
Version="7.2.0" />
  </ItemGroup>

```

```
</Project>
```

3. Delete the default `Class1.cs` file created by `dotnet` and copy the following code to a `SamplePredictorClass.cs` file in your project folder.

C#

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Management.Automation;
using System.Management.Automation.Subsystem;
using System.Management.Automation.Subsystem.Prediction;

namespace PowerShell.Sample
{
    public class SamplePredictor : ICommandPredictor
    {
        private readonly Guid _guid;

        internal SamplePredictor(string guid)
        {
            _guid = new Guid(guid);
        }

        /// <summary>
        /// Gets the unique identifier for a subsystem implementation.
        /// </summary>
        public Guid Id => _guid;

        /// <summary>
        /// Gets the name of a subsystem implementation.
        /// </summary>
        public string Name => "SamplePredictor";

        /// <summary>
        /// Gets the description of a subsystem implementation.
        /// </summary>
        public string Description => "A sample predictor";

        /// <summary>
        /// Get the predictive suggestions. It indicates the start of a
        suggestion rendering session.
        /// </summary>
        /// <param name="client">Represents the client that initiates
        the call.</param>
        /// <param name="context">The <see cref="PredictionContext"/>
        object to be used for prediction.</param>
        /// <param name="cancellationToken">The cancellation token to
        cancel the prediction.</param>
        /// <returns>An instance of <see cref="SuggestionPackage"/>.
    </returns>
```

```

        public SuggestionPackage GetSuggestion(PredictionClient client,
PredictionContext context, CancellationToken cancellationToken)
{
    string input = context.InputAst.Extent.Text;
    if (string.IsNullOrWhiteSpace(input))
    {
        return default;
    }

    return new SuggestionPackage(new List<PredictiveSuggestion>
{
    new PredictiveSuggestion(string.Concat(input, " HELLO
WORLD"))
});
}

#region "interface methods for processing feedback"

/// <summary>
/// Gets a value indicating whether the predictor accepts a
specific kind of feedback.
/// </summary>
/// <param name="client">Represents the client that initiates
the call.</param>
/// <param name="feedback">A specific type of feedback.</param>
/// <returns>True or false, to indicate whether the specific
feedback is accepted.</returns>
public bool CanAcceptFeedback(PredictionClient client,
PredictorFeedbackKind feedback) => false;

/// <summary>
/// One or more suggestions provided by the predictor were
displayed to the user.
/// </summary>
/// <param name="client">Represents the client that initiates
the call.</param>
/// <param name="session">The mini-session where the displayed
suggestions came from.</param>
/// <param name="countOrIndex">
/// When the value is greater than 0, it's the number of
displayed suggestions from the list
/// returned in <paramref name="session"/>, starting from the
index 0. When the value is
/// less than or equal to 0, it means a single suggestion from
the list got displayed, and
/// the index is the absolute value.
/// </param>
public void OnSuggestionDisplayed(PredictionClient client, uint
session, int countOrIndex) { }

/// <summary>
/// The suggestion provided by the predictor was accepted.
/// </summary>
/// <param name="client">Represents the client that initiates
the call.</param>

```

```

        ///<param name="session">Represents the mini-session where the
accepted suggestion came from.</param>
        ///<param name="acceptedSuggestion">The accepted suggestion
text.</param>
        public void OnSuggestionsAccepted(PredictionClient client, uint
session, string acceptedSuggestion) { }

        ///<summary>
        /// A command line was accepted to execute.
        /// The predictor can start processing early as needed with the
latest history.
        ///</summary>
        ///<param name="client">Represents the client that initiates
the call.</param>
        ///<param name="history">History command lines provided as
references for prediction.</param>
        public void OnCommandLineAccepted(PredictionClient client,
IReadOnlyList<string> history) { }

        ///<summary>
        /// A command line was done execution.
        ///</summary>
        ///<param name="client">Represents the client that initiates
the call.</param>
        ///<param name="commandLine">The last accepted command line.
</param>
        ///<param name="success">Shows whether the execution was
successful.</param>
        public void OnCommandLineExecuted(PredictionClient client,
string commandLine, bool success) { }

        #endregion;
    }

    ///<summary>
    /// Register the predictor on module loading and unregister it on
module un-loading.
    ///</summary>
    public class Init : IModuleAssemblyInitializer,
IModuleAssemblyCleanup
{
    private const string Identifier = "843b51d0-55c8-4c1a-8116-
f0728d419306";

    ///<summary>
    /// Gets called when assembly is loaded.
    ///</summary>
    public void OnImport()
    {
        var predictor = new SamplePredictor(Identifier);

SubsystemManager.RegisterSubsystem(SubsystemKind.CommandPredictor,
predictor);
    }
}

```

```
    /// <summary>
    /// Gets called when the binary module is unloaded.
    /// </summary>
    public void OnRemove(PSModuleInfo psModuleInfo)
    {

        SubsystemManager.UnregisterSubsystem(SubsystemKind.CommandPredictor,
        new Guid(Identifier));
    }
}
```

The following example code returns the string "HELLO WORLD" for the prediction result for all user input. Since the sample predictor doesn't process any feedback, the code doesn't implement the feedback methods from the interface. Change the prediction and feedback code to meet the needs of your predictor.

 **Note**

The list view of **PSReadLine** doesn't support multiline suggestions. Each suggestion should be a single line. If your code has a multiline suggestion, you should split the lines into separate suggestions or join the lines with a semicolon (;).

4. Run `dotnet build` to produce the assembly. You can find the compiled assembly in the `bin/Debug/net6.0` location of your project folder.

 **Note**

To ensure a responsive user experience, the **ICommandPredictor** interface has a 20ms time out for responses from the Predictors. Your predictor code must return results in less than 20ms to be displayed.

Using your predictor plugin

To try out your new predictor, open a new PowerShell 7.2 session and run the following commands:

PowerShell

```
Set-PSReadLineOption -PredictionSource HistoryAndPlugin
Import-Module .\bin\Debug\net6.0\SamplePredictor.dll
```

With the assembly is loaded in the session, you see the text "HELLO WORLD" appear as you type in the terminal. You can press **F2** to switch between the **Inline** view and the **List** view.

For more information about PSReadLine options, see [Set-PSReadLineOption](#).

You can get a list of installed predictors, using the following command:

PowerShell

```
Get-PSSubsystem -Kind CommandPredictor
```

Output

Kind	SubsystemType	IsRegistered	Implementations
CommandPredictor	ICommandPredictor	True	{SamplePredictor}

ⓘ Note

`Get-PSSubsystem` is an experimental cmdlet that was introduced in PowerShell 7.1. You must enable the `PSSubsystemPluginModel` experimental feature to use this cmdlet. For more information, see [Using Experimental Features](#).

ⓘ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[ⓘ Open a documentation issue](#)

[ⓘ Provide product feedback](#)

How to create a feedback provider

Article • 04/11/2024

PowerShell 7.4 introduced the concept of feedback providers. A feedback provider is a PowerShell module that implements the `IFeedbackProvider` interface to provide command suggestions based on user command execution attempts. The provider is triggered when there's a success or failure execution. Feedback providers use information from the success or failure to provide feedback.

Prerequisites

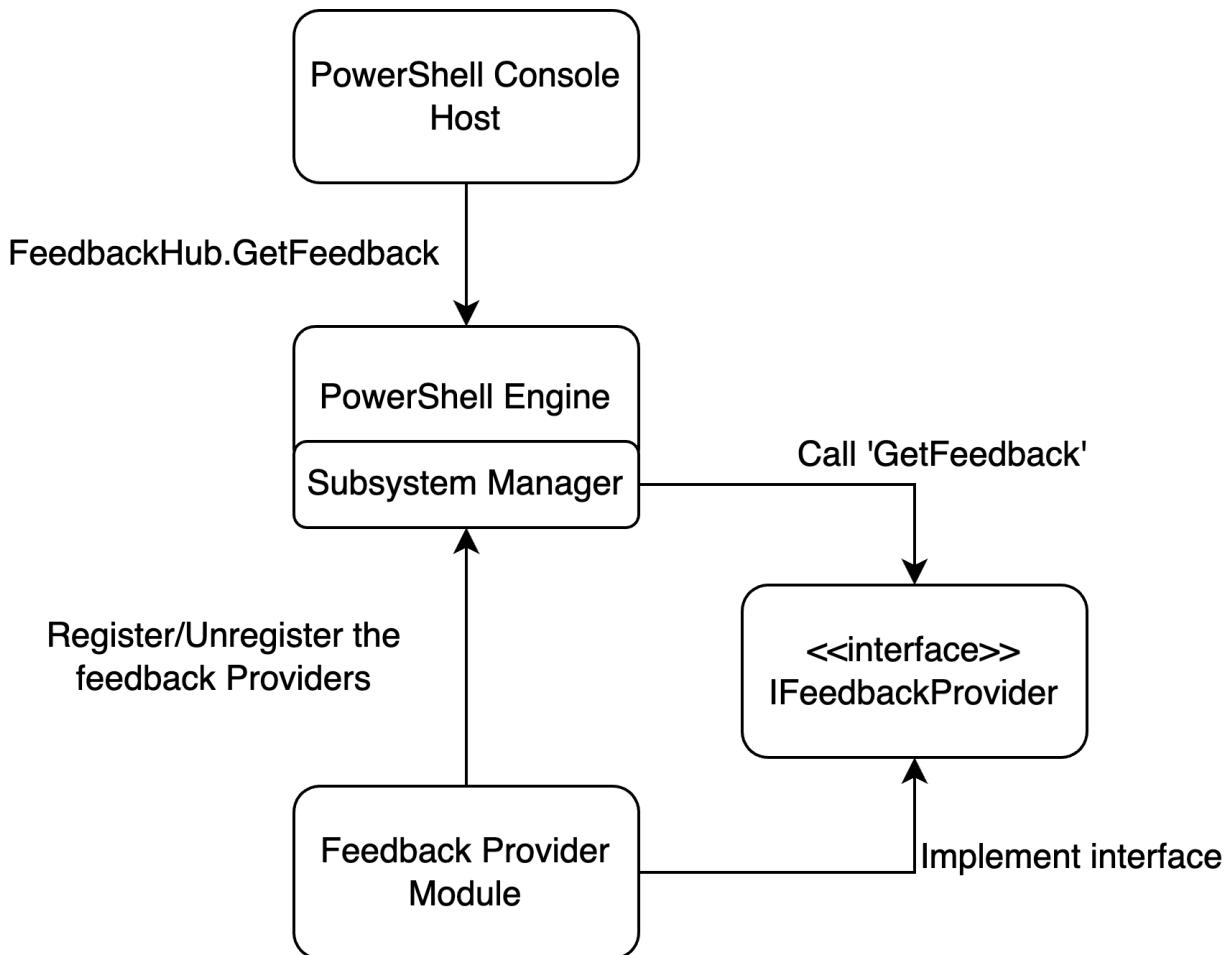
To create a feedback provider, you must satisfy the following prerequisites:

- Install PowerShell 7.4 or higher
 - You must enable the `PSFeedbackProvider` experimental feature to enable support for feedback providers and predictors. For more information, see [Using Experimental Features](#).
- Install .NET 8 SDK - 8.0.0 or higher
 - See the [Download .NET 8.0](#) page to get the latest version of the SDK.

Overview of a feedback provider

A feedback provider is a PowerShell binary module that implements the `System.Management.Automation.Subsystem.Feedback.IFeedbackProvider` interface. This interface declares the methods to get feedback based on the command line input. The feedback interface can provide suggestions based on the success or failure of the command invoked by the user. The suggestions can be anything that you want. For example, you might suggest ways to address an error or better practices, like avoiding the use of aliases. For more information, see the [What are Feedback Providers?](#) blog post.

The following diagram shows the architecture of a feedback provider:



The following examples walk you through the process of creating a simple feedback provider. Also, you can register the provider with the command predictor interface to add feedback suggestions to the command-line predictor experience. For more information about predictors, see [Using predictors in PSReadLine](#) and [How to create a command line predictor](#).

Step 1 - Create a new class library project

Use the following command to create a new project in the project directory:

```

PowerShell
dotnet new classlib --name MyFeedbackProvider
  
```

Add a package reference for the `System.Management.Automation` package to your `.csproj` file. The following example shows the updated `.csproj` file:

```

XML
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    
```

```
<TargetFramework>net8.0</TargetFramework>
<ImplicitUsings>enable</ImplicitUsings>
<Nullable>enable</Nullable>
</PropertyGroup>

<ItemGroup>
    <PackageReference Include="System.Management.Automation" Version="7.4.0-preview.3">
        <ExcludeAssets>contentFiles</ExcludeAssets>
        <PrivateAssets>All</PrivateAssets>
    </PackageReference>
</ItemGroup>
</Project>
```

ⓘ Note

You should change the version of the `System.Management.Automation` assembly to match the version of the PowerShell preview that you are targeting. The minimum version is 7.4.0-preview.3.

Step 2 - Add the class definition for your provider

Change the name of the `Class1.cs` file to match the name of your provider. This example uses `myFeedbackProvider.cs`. This file contains the two main classes that define the feedback provider. The following example shows the basic template for the class definitions.

C#

```
using System.Management.Automation;
using System.Management.Automation.Subsystem;
using System.Management.Automation.Subsystem.Feedback;
using System.Management.Automation.Subsystem.Prediction;
using System.Management.Automation.Language;

namespace myFeedbackProvider;

public sealed class myFeedbackProvider : IFeedbackProvider,
ICommandPredictor
{

}

public class Init : IModuleAssemblyInitializer, IModuleAssemblyCleanup
{
```

```
}
```

Step 3 - Implement the Init class

The `Init` class registers and unregisters the feedback provider with the subsystem manager. The `OnImport()` method runs when the binary module is being loaded. The `OnRemove()` method runs when the binary module is being removed. This example registers both the feedback provider and command predictor subsystem.

C#

```
public class Init : IModuleAssemblyInitializer, IModuleAssemblyCleanup
{
    private const string Id = "<ADD YOUR GUID HERE>";

    public void OnImport()
    {
        var feedback = new myFeedbackProvider(Id);
        SubsystemManager.RegisterSubsystem(SubsystemKind.FeedbackProvider,
feedback);
        SubsystemManager.RegisterSubsystem(SubsystemKind.CommandPredictor,
feedback);
    }

    public void OnRemove(PSModuleInfo psModuleInfo)
    {
        SubsystemManager.UnregisterSubsystem< ICommandPredictor>(new
Guid(Id));
        SubsystemManager.UnregisterSubsystem< IFeedbackProvider>(new
Guid(Id));
    }
}
```

Replace the `<ADD YOUR GUID HERE>` placeholder value with a unique Guid. You can generate a Guid using the `New-Guid` cmdlet.

PowerShell

```
New-Guid
```

The Guid is a unique identifier for your provider. The provider must have a unique Id to be registered with the subsystem.

Step 4 - Add class members and define the constructor

The following code implements the properties defined in the interfaces, adds needed class members, and creates the constructor for the `myFeedbackProvider` class.

C#

```
/// <summary>
/// Gets the global unique identifier for the subsystem implementation.
/// </summary>
private readonly Guid _guid;
public Guid Id => _guid;

/// <summary>
/// Gets the name of a subsystem implementation, this will be the name
displayed when triggered
/// </summary>
public string Name => "myFeedbackProvider";

/// <summary>
/// Gets the description of a subsystem implementation.
/// </summary>
public string Description => "This is very simple feedback provider";

/// <summary>
/// Default implementation. No function is required for a feedback provider.
/// </summary>
Dictionary<string, string>? ISubsystem.FunctionsToDefine => null;

/// <summary>
/// Gets the types of trigger for this feedback provider.
/// </summary>
/// <remarks>
/// The default implementation triggers a feedback provider by <see
/// cref="FeedbackTrigger.CommandNotFound"/> only.
/// </remarks>
public FeedbackTrigger Trigger => FeedbackTrigger.All;

/// <summary>
/// List of candidates from the feedback provider to be passed as predictor
results
/// </summary>
private List<string>? _candidates;

/// <summary>
/// PowerShell session used to run PowerShell commands that help create
suggestions.
/// </summary>
private PowerShell _powershell;

internal myFeedbackProvider(string guid)
```

```
{  
    _guid = new Guid(guid); // Save guid  
    _powershell = PowerShell.Create(); // Create PowerShell instance  
}
```

Step 5 - Create the GetFeedback() method

The `GetFeedback` method takes two parameters, `context` and `token`. The `context` parameter receives the information about the trigger so you can decide how to respond with suggestions. The `token` parameter is used for cancellation. This function returns a `FeedbackItem` containing the suggestion.

C#

```
/// <summary>  
/// Gets feedback based on the given commandline and error record.  
/// </summary>  
/// <param name="context">The context for the feedback call.</param>  
/// <param name="token">The cancellation token to cancel the operation.  
</param>  
/// <returns>The feedback item.</returns>  
public FeedbackItem? GetFeedback(FeedbackContext context, CancellationToken  
token)  
{  
    // Target describes the different kinds of triggers to activate on,  
    var target = context.Trigger;  
    var commandLine = context.CommandLine;  
    var ast = context.CommandLineAst;  
  
    // defining the header and footer variables  
    string header;  
    string footer;  
  
    // List of the actions  
    List<string>? actions = new List<string>();  
  
    // Trigger on success code goes here  
  
    // Trigger on error code goes here  
  
    return null;  
}
```

The following image shows how these fields are used in the suggestions that are displayed to the user.

```
[this is the name of feedback provider]
  this is the header
    > these
    > are
    > actions

  this is the footer
```

Create suggestions for a Success trigger

For a successful invocation, we want to expand any aliases used in the last execution. Using the `CommandLineAst`, we identify any aliased commands and create a suggestion to use the fully qualified command name instead.

C#

```
// Trigger on success
if (target == FeedbackTrigger.Success)
{
    // Getting the commands from the AST and only finding those that are
    Commands
    var astCmds = ast.FindAll((cAst) => cAst is CommandAst, true);

    // Inspect each of the commands
    foreach(var command in astCmds)
    {

        // Get the command name
        var aliasedCmd = ((CommandAst) command).GetCommandName();

        // Check if its an alias or not, if so then add it to the list of
        actions
        if(TryGetAlias(aliasedCmd, out string commandString))
        {
            actions.Add($"{aliasedCmd} --> {commandString}");
        }
    }

    // If no alias was found return null
    if(actions.Count == 0)
    {
        return null;
    }

    // If aliases are found, set the header to a description and return a
    new FeedbackItem.
    header = "You have used an aliased command:";
    // Copy actions to _candidates for the predictor
    _candidates = actions;
```

```
        return new FeedbackItem(header, actions);
    }
```

Implement the TryGetAlias() method

The `TryGetAlias()` method is a private helper function that returns a boolean value to indicate whether the command is an alias. In the class constructor, we created a PowerShell instance that we can use to run PowerShell commands. The `TryGetAlias()` method uses this PowerShell instance to invoke the `GetCommand` method to determine if the command is an alias. The `AliasInfo` object returned by `GetCommand` contains full name of the aliased command.

C#

```
/// <summary>
/// Checks if a command is an alias.
/// </summary>
/// <param name="command">The command to check if alias</param>
/// <param name="targetCommand">The referenced command by the aliased
/// command</param>
/// <returns>True if an alias and false if not</returns>
private bool TryGetAlias(string command, out string targetCommand)
{
    // Create PowerShell runspace as a session state proxy to run GetCommand
    // and check
    // if its an alias
    AliasInfo? pwshAliasInfo =
        _powershell.Runspace.SessionStateProxy.InvokeCommand.GetCommand(command,
        CommandTypes.Alias) as AliasInfo;

    // if its null then it is not an aliased command so just return false
    if(pwshAliasInfo is null)
    {
        targetCommand = String.Empty;
        return false;
    }

    // Set targetCommand to referenced command name
    targetCommand = pwshAliasInfo.ReferencedCommand.Name;
    return true;
}
```

Create suggestions for a Failure trigger

When a command execution fails, we want to suggest that the user `Get-Help` to get more information about how to use the command.

C#

```
// Trigger on error
if (target == FeedbackTrigger.Error)
{
    // Gets the command that caused the error.
    var erroredCommand = context.LastError?.InvocationInfo.MyCommand;
    if (erroredCommand is null)
    {
        return null;
    }

    header = $"You have triggered an error with the command
{erroredCommand}. Try using the following command to get help:";

    actions.Add($"Get-Help {erroredCommand}");
    footer = $"You can also check online documentation at
https://learn.microsoft.com/en-us/powershell/module/?term={erroredCommand}";

    // Copy actions to _candidates for the predictor
    _candidates = actions;
    return new FeedbackItem(header, actions, footer,
FeedbackDisplayLayout.Portrait);
}
```

Step 6 - Send suggestions to the command line predictor

Another way your feedback provider can enhance the user experience is to provide command suggestions to the **ICommandPredictor** interface. For more information about creating a command line predictor, see [How to create a command line predictor](#).

The following code implements the methods necessary from the **ICommandPredictor** interface to add predictor behavior to your feedback provider.

- `CanAcceptFeedback()` - This method returns a Boolean value that indicates whether the predictor accepts a specific type of feedback.
- `GetSuggestion()` - This method returns a `SuggestionPackage` object that contains the suggestions to be displayed by the predictor.
- `OnCommandLineAccepted()` - This method is called when a command line is accepted to execute.

C#

```
/// <summary>
/// Gets a value indicating whether the predictor accepts a specific kind of
/// feedback.
```

```

    ///> </summary>
    ///> <param name="client">Represents the client that initiates the call.</param>
    ///> <param name="feedback">A specific type of feedback.</param>
    ///> <returns>True or false, to indicate whether the specific feedback is accepted.</returns>
    public bool CanAcceptFeedback(PredictionClient client, PredictorFeedbackKind feedback)
    {
        return feedback switch
        {
            PredictorFeedbackKind.CommandLineAccepted => true,
            _ => false,
        };
    }

    ///> <summary>
    ///> Get the predictive suggestions. It indicates the start of a suggestion rendering session.
    ///> </summary>
    ///> <param name="client">Represents the client that initiates the call.</param>
    ///> <param name="context">The <see cref="PredictionContext"/> object to be used for prediction.</param>
    ///> <param name="cancellationToken">The cancellation token to cancel the prediction.</param>
    ///> <returns>An instance of <see cref="SuggestionPackage"/>.</returns>
    public SuggestionPackage GetSuggestion(
        PredictionClient client,
        PredictionContext context,
        CancellationToken cancellationToken)
    {
        if (_candidates is not null)
        {
            string input = context.InputAst.Extent.Text;
            List<PredictiveSuggestion>? result = null;

            foreach (string c in _candidates)
            {
                if (c.StartsWith(input, StringComparison.OrdinalIgnoreCase))
                {
                    result ??= new List<PredictiveSuggestion>(_candidates.Count);
                    result.Add(new PredictiveSuggestion(c));
                }
            }

            if (result is not null)
            {
                return new SuggestionPackage(result);
            }
        }

        return default;
    }
}

```

```

/// <summary>
/// A command line was accepted to execute.
/// The predictor can start processing early as needed with the latest
/// history.
/// </summary>
/// <param name="client">Represents the client that initiates the call.
</param>
/// <param name="history">History command lines provided as references for
/// prediction.</param>
public void OnCommandLineAccepted(PredictionClient client,
IReadOnlyList<string> history)
{
    // Reset the candidate state once the command is accepted.
    _candidates = null;
}

```

Step 7 - Build the feedback provider

Now you are ready to build and begin using your feedback provider! To build the project, run the following command:

PowerShell

```
dotnet build
```

This command create the PowerShell module as a DLL file in the following path of your project folder: `bin/Debug/net8.0/myFeedbackProvider`

You may run into the error `error NU1101: Unable to find package`

`System.Management.Automation`. when building on Windows machines. To fix this add a `nuget.config` file to your project directory and add the following:

YAML

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <packageSources>
        <clear />
        <add key="nuget.org" value="https://api.nuget.org/v3/index.json" />
    </packageSources>
    <disabledPackageSources>
        <clear />
    </disabledPackageSources>
</configuration>

```

Using a feedback provider

To test your new feedback provider, import the compiled module into your PowerShell session. This can be done by importing the folder described after building has succeeded:

```
PowerShell
```

```
Import-Module ./bin/Debug/net8.0/myFeedbackProvider
```

Once you're satisfied with your module, you should create a module manifest, publish it to the PowerShell Gallery, and install it in your `$Env:PSModulePath`. For more information, see [How to create a module manifest](#). You can add the `Import-Module` command to your `$PROFILE` script so the module is available in PowerShell session.

You can get a list of installed feedback providers, using the following command:

```
PowerShell
```

```
Get-PSSubsystem -Kind FeedbackProvider
```

```
Output
```

Kind	SubsystemType	IsRegistered	Implementations
---	-----	-----	-----
FeedbackProvider	IFeedbackProvider	True	{general}

ⓘ Note

`Get-PSSubsystem` is an experimental cmdlet that was introduced in PowerShell 7.1. You must enable the `PSSubsystemPluginModel` experimental feature to use this cmdlet. For more information, see [Using Experimental Features](#).

The following screenshot shows some example suggestions from the new provider.

```
PS C:\Users\stevenbucher.REDMOND> Get-ChildItem -badparameter
Get-ChildItem: A parameter cannot be found that matches parameter name 'badparameter'.

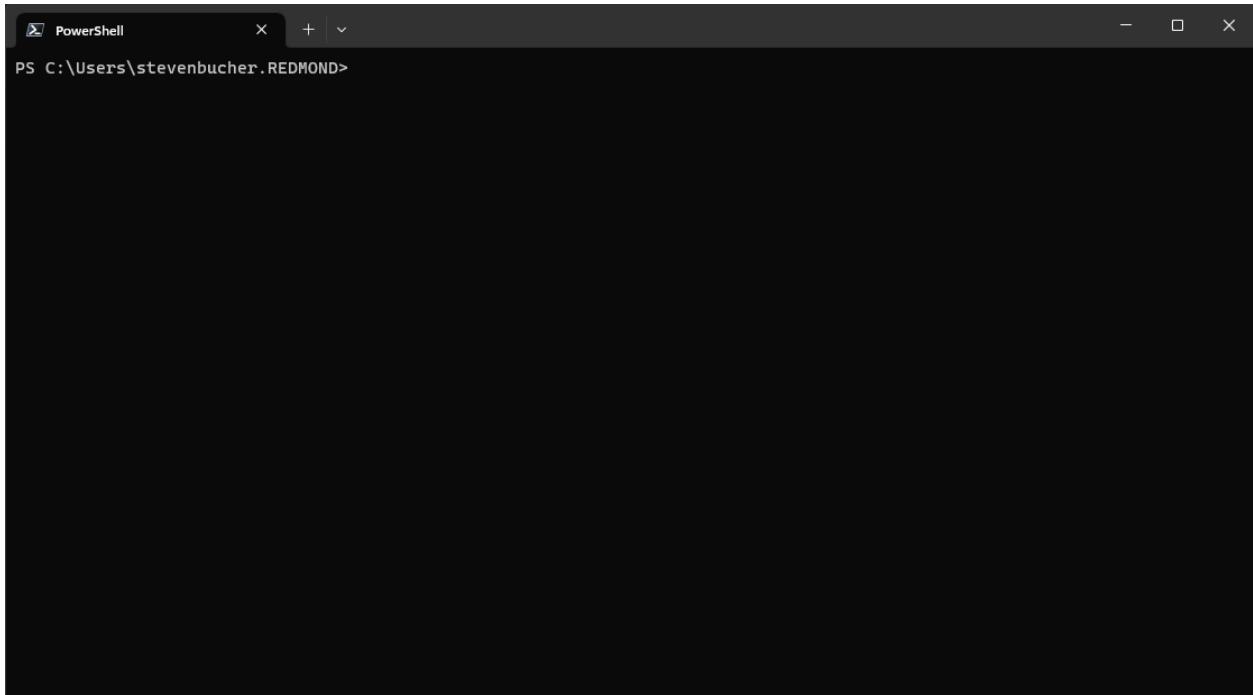
[myFeedbackProvider]
You have triggered an error with the command Get-ChildItem. Try using the following command to get help:
  > Get-Help Get-ChildItem

You can also check online documentation at https://learn.microsoft.com/en-us/powershell/module/?term=Get-ChildItem

PS C:\Users\stevenbucher.REDMOND> oh -In "Hello World"
Hello World

[myFeedbackProvider]
You have used an aliased command:
  > oh --> Out-Host
```

The following is a GIF showing how the predictor integration works from the new provider.



Other feedback providers

We have created other feedback provider that can be used as a good reference for deeper examples.

command-not-found

The `command-not-found` feedback provider utilizes the `command-not-found` utility tool on Linux systems to provide suggestions when native commands are attempted to run but are missing. You can find the code in the [GitHub Repository](#) or can download for yourself on the [PowerShell Gallery](#).

PowerShell Adapter

The `Microsoft.PowerShell.PowerShellAdapter` is a feedback provider that helps you convert text outputs from native commands into PowerShell objects. It detects "adapters" on your system and suggests you to use them when you use the native command. You can learn more about PowerShell Adapters from, [PowerShell Adapter Feedback Provider](#) blog post. You can also find the code in the [GitHub Repository](#) or can download for yourself on the [PowerShell Gallery](#).

Appendix - Full implementation code

The following code combines the previous examples into the full implementation of the provider class.

C#

```
using System.Management.Automation;
using System.Management.Automation.Subsystem;
using System.Management.Automation.Subsystem.Feedback;
using System.Management.Automation.Subsystem.Prediction;
using System.Management.Automation.Language;

namespace myFeedbackProvider;

public sealed class myFeedbackProvider : IFeedbackProvider,
ICommandPredictor
{
    /// <summary>
    /// Gets the global unique identifier for the subsystem implementation.
    /// </summary>
    private readonly Guid _guid;
    public Guid Id => _guid;

    /// <summary>
    /// Gets the name of a subsystem implementation, this will be the name
    displayed when triggered
    /// </summary>
    public string Name => "myFeedbackProvider";

    /// <summary>
    /// Gets the description of a subsystem implementation.
    /// </summary>
    public string Description => "This is very simple feedback provider";

    /// <summary>
    /// Default implementation. No function is required for a feedback
    provider.
    /// </summary>
    Dictionary<string, string>? ISubsystem.FunctionsToDefine => null;

    /// <summary>
    /// Gets the types of trigger for this feedback provider.
    /// </summary>
```

```

    ///</summary>
    ///<remarks>
    /// The default implementation triggers a feedback provider by <see
    cref="FeedbackTrigger.CommandNotFound"/> only.
    ///</remarks>
    public FeedbackTrigger Trigger => FeedbackTrigger.All;

    ///<summary>
    /// List of candidates from the feedback provider to be passed as
    predictor results
    ///</summary>
    private List<string>? _candidates;

    ///<summary>
    /// PowerShell session used to run PowerShell commands that help create
    suggestions.
    ///</summary>
    private PowerShell _powershell;

    // Constructor
    internal myFeedbackProvider(string guid)
    {
        _guid = new Guid(guid); // Save guid
        _powershell = PowerShell.Create(); // Create PowerShell instance
    }

    #region IFeedbackProvider
    ///<summary>
    /// Gets feedback based on the given commandline and error record.
    ///</summary>
    ///<param name="context">The context for the feedback call.</param>
    ///<param name="token">The cancellation token to cancel the operation.
    </param>
    ///<returns>The feedback item.</returns>
    public FeedbackItem? GetFeedback(FeedbackContext context,
    CancellationToken token)
    {
        // Target describes the different kinds of triggers to activate on,
        var target = context.Trigger;
        var commandLine = context.CommandLine;
        var ast = context.CommandLineAst;

        // defining the header and footer variables
        string header;
        string footer;

        // List of the actions
        List<string>? actions = new List<string>();

        // Trigger on success
        if (target == FeedbackTrigger.Success)
        {
            // Getting the commands from the AST and only finding those that
            are Commands
            var astCmds = ast.FindAll((cAst) => cAst is CommandAst, true);

```

```

// Inspect each of the commands
foreach(var command in astCmds)
{
    // Get the command name
    var aliasedCmd = ((CommandAst) command).GetCommandName();

    // Check if its an alias or not, if so then add it to the
list of actions
    if(TryGetAlias(aliasedCmd, out string commandString))
    {
        actions.Add($"{aliasedCmd} --> {commandString}");
    }
}

// If no alias was found return null
if(actions.Count == 0)
{
    return null;
}

// If aliases are found, set the header to a description and
return a new FeedbackItem.
header = "You have used an aliased command:";
// Copy actions to _candidates for the predictor
_candidates = actions;

return new FeedbackItem(header, actions);
}

// Trigger on error
if (target == FeedbackTrigger.Error)
{
    // Gets the command that caused the error.
    var erroredCommand =
context.LastError?.InvocationInfo.MyCommand;
    if (erroredCommand is null)
    {
        return null;
    }

    header = $"You have triggered an error with the command
{erroredCommand}. Try using the following command to get help:";

    actions.Add($"Get-Help {erroredCommand}");
    footer = $"You can also check online documentation at
https://learn.microsoft.com/en-us/powershell/module/?term={erroredCommand}";

    // Copy actions to _candidates for the predictor
    _candidates = actions;
    return new FeedbackItem(header, actions, footer,
FeedbackDisplayLayout.Portal);
}
return null;

```

```

}

/// <summary>
/// Checks if a command is an alias.
/// </summary>
/// <param name="command">The command to check if alias</param>
/// <param name="targetCommand">The referenced command by the aliased
command</param>
/// <returns>True if an alias and false if not</returns>
private bool TryGetAlias(string command, out string targetCommand)
{
    // Create PowerShell runspace as a session state proxy to run
GetCommand and check
    // if its an alias
    AliasInfo? pwshAliasInfo =
.Runspace.SessionStateProxy.InvokeCommand(command,
CommandTypes.Alias) as AliasInfo;

    // if its null then it is not an aliased command so just return
false
    if(pwshAliasInfo is null)
    {
        targetCommand = String.Empty;
        return false;
    }

    // Set targetCommand to referenced command name
    targetCommand = pwshAliasInfo.ReferencedCommand.Name;
    return true;
}
#endregion IFeedbackProvider

#region ICommandPredictor

/// <summary>
/// Gets a value indicating whether the predictor accepts a specific
kind of feedback.
/// </summary>
/// <param name="client">Represents the client that initiates the call.
</param>
/// <param name="feedback">A specific type of feedback.</param>
/// <returns>True or false, to indicate whether the specific feedback is
accepted.</returns>
public bool CanAcceptFeedback(PredictionClient client,
PredictorFeedbackKind feedback)
{
    return feedback switch
    {
        PredictorFeedbackKind.CommandLineAccepted => true,
        _ => false,
    };
}

/// <summary>

```

```

    /// Get the predictive suggestions. It indicates the start of a
    suggestion rendering session.
    /// </summary>
    /// <param name="client">Represents the client that initiates the call.
</param>
    /// <param name="context">The <see cref="PredictionContext"/> object to
    be used for prediction.</param>
    /// <param name="cancellationToken">The cancellation token to cancel the
    prediction.</param>
    /// <returns>An instance of <see cref="SuggestionPackage"/>.</returns>
    public SuggestionPackage GetSuggestion(
        PredictionClient client,
        PredictionContext context,
        CancellationToken cancellationToken)
    {
        if (_candidates is not null)
        {
            string input = context.InputAst.Extent.Text;
            List<PredictiveSuggestion>? result = null;

            foreach (string c in _candidates)
            {
                if (c.StartsWith(input, StringComparison.OrdinalIgnoreCase))
                {
                    result ??= new List<PredictiveSuggestion>(_candidates.Count);
                    result.Add(new PredictiveSuggestion(c));
                }
            }

            if (result is not null)
            {
                return new SuggestionPackage(result);
            }
        }
    }

    return default;
}

/// <summary>
/// A command line was accepted to execute.
/// The predictor can start processing early as needed with the latest
history.
/// </summary>
/// <param name="client">Represents the client that initiates the call.
</param>
    /// <param name="history">History command lines provided as references
    for prediction.</param>
    public void OnCommandLineAccepted(PredictionClient client,
    IReadOnlyList<string> history)
    {
        // Reset the candidate state once the command is accepted.
        _candidates = null;
    }

```

```
#endregion;  
}  
  
public class Init : IModuleAssemblyInitializer, IModuleAssemblyCleanup  
{  
    private const string Id = "<ADD YOUR GUID HERE>";  
  
    public void OnImport()  
    {  
        var feedback = new myFeedbackProvider(Id);  
        SubsystemManager.RegisterSubsystem(SubsystemKind.FeedbackProvider,  
feedback);  
        SubsystemManager.RegisterSubsystem(SubsystemKind.CommandPredictor,  
feedback);  
    }  
  
    public void OnRemove(PSModuleInfo psModuleInfo)  
    {  
        SubsystemManager.UnregisterSubsystem< ICommandPredictor>(new  
Guid(Id));  
        SubsystemManager.UnregisterSubsystem< IFeedbackProvider>(new  
Guid(Id));  
    }  
}
```

Create XML-based help using PlatyPS

Article • 12/12/2024

When creating a PowerShell module, it's always recommended that you include detailed help for the cmdlets you create. If your cmdlets are implemented in compiled code, you must use the XML-based help. This XML format is known as the Microsoft Assistance Markup Language (MAML).

The legacy PowerShell SDK documentation covers the details of creating help for PowerShell cmdlets packaged into modules. However, PowerShell doesn't provide any tools for creating the XML-based help. The SDK documentation explains the structure of MAML help, but leaves you the task of creating the complex, and deeply nested, MAML content by hand.

This is where the [PlatyPS](#) module can help.

What is PlatyPS?

PlatyPS is an [open-source](#) tool that started as a *hackathon* project to make the creation and maintenance of MAML easier. PlatyPS documents the syntax of parameter sets and the individual parameters for each cmdlet in your module. PlatyPS creates structured [Markdown](#) files that contain the syntax information. It can't create descriptions or provide examples.

PlatyPS creates placeholders for you to fill in descriptions and examples. After adding the required information, PlatyPS compiles the Markdown files into MAML files. PowerShell's help system also allows for plain-text conceptual help files (about topics). PlatyPS has a cmdlet to create a structured Markdown template for a new *about* file, but these `about_*.help.txt` files must be maintained manually.

You can include the MAML and Text help files with your module. You can also use PlatyPS to compile the help files into a CAB package that can be hosted for updateable help.

Get started using PlatyPS

First you must install PlatyPS from the PowerShell Gallery.

```
PowerShell

# Install using PowerShellGet 2.x
Install-Module platyps -Force

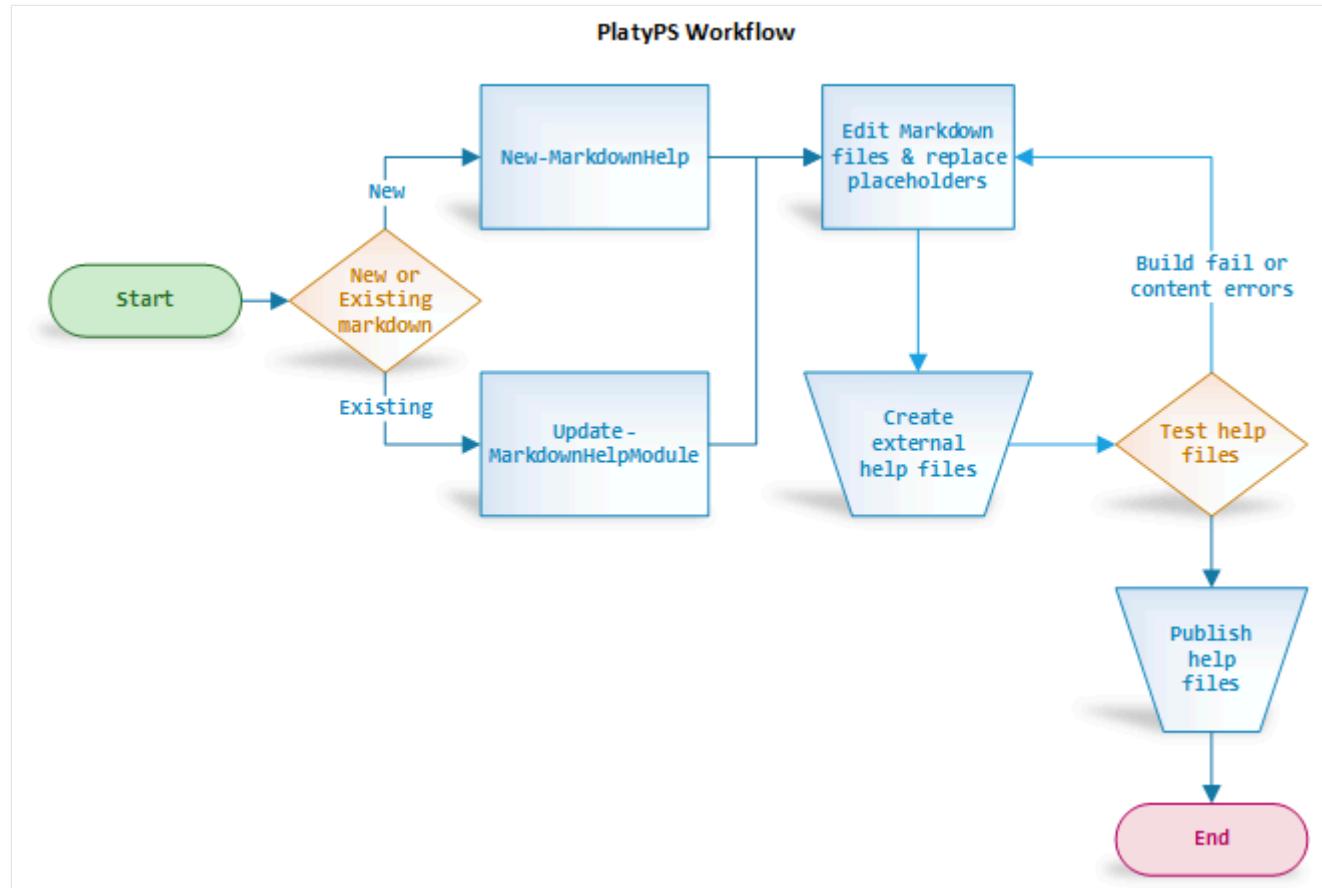
# Install using PSResourceGet 1.x
Install-PSResource platyps -Reinstall
```

After installing PlatyPS, import the module into your session.

PowerShell

```
Import-Module platyp
```

The following flowchart outlines the process for creating or updating PowerShell reference content.



Create Markdown content for a PowerShell module

1. Import your new module into the session. Repeat this step for each module you need to document.

Run the following command to import your modules:

PowerShell

```
Import-Module <your module name>
```

2. Use PlatyPS to generate Markdown files for your module page and all associated cmdlets within the module. Repeat this step for each module you need to document.

PowerShell

```
$OutputFolder = <output path>
$parameters = @{
    Module = <ModuleName>
    OutputFolder = $OutputFolder
    AlphabeticParamsOrder = $true
    WithModulePage = $true
    ExcludeDontShow = $true
    Encoding = [System.Text.Encoding]::UTF8
}
New-MarkdownHelp @parameters

New-MarkdownAboutHelp -OutputFolder $OutputFolder -AboutName "topic_name"
```

If the output folder doesn't exist, `New-MarkdownHelp` creates it. In this example, `New-MarkdownHelp` creates a Markdown file for each cmdlet in the module. It also creates the *module page* in a file named `<ModuleName>.md`. This module page contains a list of the cmdlets contained in the module and placeholders for the *Synopsis* description. The metadata in the module page comes from the module manifest and is used by PlatyPS to create the `HelpInfo` XML file (as explained [below](#)).

`New-MarkdownAboutHelp` creates a new *about* file named `about_topic_name.md`.

For more information, see [New-MarkdownHelp](#) and [New-MarkdownAboutHelp](#).

Update existing Markdown content when the module changes

PlatyPS can also update existing Markdown files for a module. Use this step to update existing modules that have new cmdlets, new parameters, or parameters that have changed.

1. Import your new module into the session. Repeat this step for each module you need to document.

Run the following command to import your modules:

PowerShell

```
Import-Module <your module name>
```

2. Use PlatyPS to update Markdown files for your module landing page and all associated cmdlets within the module. Repeat this step for each module you need to document.

PowerShell

```
$parameters = @{
    Path = <folder with Markdown>
    RefreshModulePage = $true
    AlphabeticParamsOrder = $true
    UpdateInputOutput = $true
    ExcludeDontShow = $true
    LogPath = <path to store log file>
    Encoding = [System.Text.Encoding]::UTF8
}
Update-MarkdownHelpModule @parameters
```

`Update-MarkdownHelpModule` updates the cmdlet and module Markdown files in the specified folder. It doesn't update the `about_*.md` files. The module file (`ModuleName.md`) receives any new **Synopsis** text that has been added to the cmdlet files. Updates to cmdlet files include the following change:

- New parameter sets
- New parameters
- Updated parameter metadata
- Updated input and output type information

For more information, see [Update-MarkdownHelpModule](#).

Edit the new or updated Markdown files

PlatyPS documents the syntax of the parameter sets and the individual parameters. It can't create descriptions or provide examples. The specific areas where content is needed are contained in curly braces. For example: `{{ Fill in the Description }}`

```
File Edit Selection View Go Run Terminal Help New-CMAdministrativeUserPermission.md - ConfigurationManager - Visual Studio Code
New-CMAdministrativeUserPermission.md (deleted) X
New-CMAdministrativeUserPermission.md > # New-CMAdministrativeUserPermission > ## EXAMPLES > ### Example 1
35 [-CollectionName <String[]>] [-Collection <IResultObject[]>] [-DisableWildcardHandling]
36 [-ForceWildcardHandling] [<CommonParameters>]
37
38
39 ## DESCRIPTION
40 {{ Fill in the Description }}
41
42 > [!NOTE]
43 Configuration Manager cmdlets must be run from the Configuration Manager site drive.
44 The examples in this article use the site name **XYZ**. For more information, see the
45 [getting started](/powershell/sccm/overview) documentation.
46
47 ## EXAMPLES
48
49 #### Example 1
50 powershell
51 PS XYZ:\> {{ Add example code here }}
52 ...
53
54 {{ Add example description here }}
55
56 ## PARAMETERS
57
58 #### -Collection
59 {{ Fill Collection Description }}
60
61 ````yaml
62 Type: IResultObject[]
63 Parameter Sets: (All)
64 Aliases: Collections
65
66 Required: False
67 Position: Named
68 Default Value: None
69 Accept Pipeline Input: False
70 Accept Wildcard Characters: False
71 ````
```

You need to add a synopsis, a description of the cmdlet, descriptions for each parameter, and at least one example.

For detailed information about writing PowerShell content, see the following articles:

- [PowerShell style guide](#)
- [Editing reference articles](#)

(!) Note

PlatyPS has a specific schema that's used for cmdlet reference. That schema only allows certain Markdown blocks in specific sections of the document. If you put content in the wrong location, the PlatyPS build step fails. For more information, see the [schema](#) documentation in the PlatyPS repository. For a complete example of well-formed cmdlet reference, see [Get-Item](#).

After providing the required content for each of your cmdlets, you need to make sure that you update the module landing page. Verify your module has the correct [Module Guid](#) and

Download Help Link values in the YAML metadata of the <module-name>.md file. Add any missing metadata.

Also, you may notice that some cmdlets may be missing a **Synopsis** (*short description*). The following command updates the module landing page with your **Synopsis** description text. Open the module landing page to verify the descriptions.

PowerShell

```
Update-MarkdownHelpModule -Path <full path output folder> -RefreshModulePage
```

Now that you have entered all the content, you can create the MAML help files that are displayed by Get-Help in the PowerShell console.

Create the external help files

This step creates the MAML help files that are displayed by Get-Help in the PowerShell console.

To build the MAML files, run the following command:

PowerShell

```
New-ExternalHelp -Path <folder with MDs> -OutputPath <output help folder>
```

New-ExternalHelp converts all cmdlet Markdown files into one (or more) MAML files. About files are converted to plain-text files with the following name format:

about_topic_name.help.txt. The Markdown content must meet the requirement of the PlatyPS schema. New-ExternalHelp exits with errors when the content doesn't follow the schema. You must edit the files to fix the schema violations.

✖ Caution

PlatyPS does a poor job converting the about_*.md files to plain text. You must use very simple Markdown to get acceptable results. You may want to maintain the files in plain-text about_topic_name.help.txt format, rather than allowing PlatyPS to convert them.

Once this step is complete, you will see *-help.xml and about_*.help.txt files in the target output folder.

For more information, see [New-ExternalHelp](#)

Test the compiled help files

You can verify the content with the [Get-HelpPreview](#) cmdlet:

```
PowerShell
```

```
Get-HelpPreview -Path "<ModuleName>-Help.xml"
```

The cmdlet reads the compiled MAML file and outputs the content in the same format you would receive from [Get-Help](#). This allows you to inspect the results to verify that the Markdown files compiled correctly and produce the desired results. If you find an error, re-edit the Markdown files and recompile the MAML.

Now you are ready to publish your help files.

Publishing your help

Now that you have compiled the Markdown files into PowerShell help files, you are ready to make the files available to users. There are two options for providing help in the PowerShell console.

- Package the help files with the module
- Create an updateable help package that users install with the [Update-Help](#) cmdlet

Packaging help with the module

The help files can be packaged with your module. See [Writing Help for Modules](#) for details of the folder structure. You should include the list of Help files in the value of the **FileList** key in the module manifest.

Creating an updateable help package

At a high level, the steps to create updateable help include:

1. Find an internet site to host your help files
2. Add a **HelpInfoURI** key to your module manifest
3. Create a HelpInfo XML file
4. Create CAB files
5. Upload your files

For more information, see [Supporting Updateable Help: Step-by-step](#).

PlatyPS assists you with some of these steps.

The **HelpInfoURI** is a URL that points to location where your help files are hosted on the internet. This value is configured in the module manifest. `Update-Help` reads the module manifest to get this URL and download the updateable help content. This URL should only point to the folder location and not to individual files. `Update-Help` constructs the filenames to download based on other information from the module manifest and the HelpInfo XML file.

Important

The **HelpInfoURI** must end with a forward-slash character (/). Without that character, `Update-Help` can't construct the correct file paths to download the content. Also, most HTTP-based file services are case-sensitive. It's important that the module metadata in the HelpInfo XML file contains the proper letter case.

The `New-ExternalHelp` cmdlet creates the HelpInfo XML file in the output folder. The HelpInfo XML file is built using YAML metadata contained in the module Markdown files (`ModuleName.md`).

The `New-ExternalHelpCab` cmdlet creates ZIP and CAB files containing the MAML and `about_*.help.txt` files you compiled. CAB files are compatible with all versions of PowerShell. PowerShell 6 and higher can use ZIP files.

PowerShell

```
$helpCabParameters = @{
    CabFilesFolder = $MamlOutputFolder
    LandingPagePath = $LandingPage
    OutputFolder = $CabOutputFolder
}
New-ExternalHelpCab @helpCabParameters
```

After creating the ZIP and CAB files, upload the ZIP, CAB, and HelpInfo XML files to your HTTP file server. Put the files in the location indicated by the **HelpInfoURI**.

For more information, see [New-ExternalHelpCab](#).

Other publishing options

Markdown is a versatile format that's easy to transform to other formats for publishing. Using a tool like [Pandoc](#) ↗, you can convert your Markdown help files to many different document formats, including plain text, HTML, PDF, and Office document formats.

Also, the cmdlets [ConvertFrom-Markdown](#) and [Show-Markdown](#) in PowerShell 6 and higher can be used to convert Markdown to HTML or create a colorful display in the PowerShell console.

Known issues

PlatyPS is very sensitive to the [schema](#) for the structure of the Markdown files it creates and compiles. It's very easy write valid Markdown that violates this schema. For more information, see the [PowerShell style guide](#) and [Editing reference articles](#).

Windows PowerShell Language Specification 3.0

Article • 01/08/2025

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

1. Introduction

PowerShell is a command-line *shell* and scripting language, designed especially for system administrators.

Most shells operate by executing a command or utility in a new process, and presenting the results to the user as text. These shells also have commands that are built into the shell and run in the shell process. Because there are few built-in commands, many utilities have been created to supplement them. PowerShell is very different. Instead of processing text, the shell processes objects. PowerShell also includes a large set of built-in commands with each having a consistent interface and these can work with user-written commands.

An *object* is a data entity that has *properties* (i.e., characteristics) and *methods* (i.e., actions that can be performed on the object). All objects of the same type have the same base set of properties and methods, but each *instance* of an object can have different property values.

A major advantage of using objects is that it is much easier to *pipeline* commands; that is, to write the output of one command to another command as input. (In a traditional command-line environment, the text output from one command needs to be manipulated to meet the input format of another.)

PowerShell includes a very rich scripting language that supports constructs for looping, conditions, flow-control, and variable assignment. This language has syntax features and keywords similar to those used in the C# programming language ([§C](#)).

There are four kinds of commands in PowerShell: scripts, functions and methods, cmdlets, and native commands.

- A file of commands is called a *script*. By convention, a script has a filename extension of .ps1. The top-most level of a PowerShell program is a script, which, in turn, can invoke other commands.
- PowerShell supports modular programming via named procedures. A procedure written in PowerShell is called a *function*, while an external procedure made available by the execution environment (and typically written in some other language) is called a *method*.
- A *cmdlet* (pronounced "command-let") is a simple, single-task command-line tool. Although a cmdlet can be used on its own, the full power of cmdlets is realized when they are used in combination to perform complex tasks.
- A *native command* is part of the host environment.

Each time the PowerShell runtime environment begins execution, it begins what is called a *session*. Commands then execute within the context of that session.

This specification defines the PowerShell language, the built-in cmdlets, and the use of objects via the pipeline.

Unlike most shells, which accept and return text, Windows PowerShell is built on top of the .NET Framework common language runtime (CLR) and the .NET Framework, and accepts and returns .NET Framework objects.

2. Lexical Structure

Article • 04/23/2024

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

2.1 Grammars

This specification shows the syntax of the PowerShell language using two grammars. The *lexical grammar* ([§B.1](#)) shows how Unicode characters are combined to form line terminators, comments, white space, and tokens. The *syntactic grammar* ([§B.2](#)) shows how the tokens resulting from the lexical grammar are combined to form PowerShell scripts.

For convenience, fragments of these grammars are replicated in appropriate places throughout this specification.

Any use of the characters 'a' through 'z' in the grammars is case insensitive. This means that letter case in variables, aliases, function names, keywords, statements, and operators is ignored. However, throughout this specification, such names are written in lowercase, except for some automatic and preference variables.

2.2 Lexical analysis

2.2.1 Scripts

Syntax:

Tip

The `~opt~` notation in the syntax definitions indicates that the lexical entity is optional in the syntax.

Syntax

```
input:
    input-elements~opt~    signature-block~opt~

input-elements:
    input-element
    input-elements    input-element

input-element:
    whitespace
    comment
    token

signature-block:
    signature-begin    signature    signature-end

signature-begin:
    new-line-character    # SIG # Begin signature block    new-line-character

signature:
    base64 encoded signature blob in multiple single-line-comments

signature-end:
    new-line-character    # SIG # End signature block    new-line-character
```

Description:

The input source stream to a PowerShell translator is the *input* in a script, which contains a sequence of Unicode characters. The lexical processing of this stream involves the reduction of those characters into a sequence of tokens, which go on to become the input of syntactic analysis.

A script is a group of PowerShell commands stored in a *script-file*. The script itself has no name, per se, and takes its name from its source file. The end of that file indicates the end of the script.

A script may optionally contain a digital signature. A host environment is not required to process any text that follows a signature or anything that looks like a signature. The creation and use of digital signatures are not covered by this specification.

2.2.2 Line terminators

Syntax:

```
Syntax

new-line-character:
    Carriage return character (U+000D)
    Line feed character (U+000A)
    Carriage return character (U+000D) followed by line feed character
(U+000A)

new-lines:
    new-line-character
    new-lines new-line-character
```

Description:

The presence of *new-line-characters* in the input source stream divides it into lines that can be used for such things as error reporting and the detection of the end of a single-line comment.

A line terminator can be treated as white space ([§2.2.4](#)).

2.2.3 Comments

Syntax:

```
Syntax

comment:
    single-line-comment
    requires-comment
    delimited-comment

single-line-comment:
    # input-characters~opt~

input-characters:
    input-character
    input-characters input-character

input-character:
    Any Unicode character except a new-line-character
```

```

requires-comment:
    #Requires whitespace command-arguments

dash:
    - (U+002D)
        EnDash character (U+2013)
        EmDash character (U+2014)
        Horizontal bar character (U+2015)

dashdash:
    dash dash

delimited-comment:
    < # delimited-comment-text~opt~ hashes >

delimited-comment-text:
    delimited-comment-section
    delimited-comment-text delimited-comment-section

delimited-comment-section:
    >
    hashes~opt~ not-greater-than-or-hash

hashes:
    #
    hashes #

not-greater-than-or-hash:
    Any Unicode character except > or #

```

Description:

Source code can be annotated by the use of *comments*.

A *single-line-comment* begins with the character `#` and ends with a *new-line-character*.

A *delimited-comment* begins with the character pair `<#` and ends with the character pair `#>`. It can occur as part of a source line, as a whole source line, or it can span any number of source lines.

A comment is treated as white space.

The productions above imply that

- Comments do not nest.
- The character sequences `<#` and `#>` have no special meaning in a single-line comment.
- The character `#` has no special meaning in a delimited comment.

The lexical grammar implies that comments cannot occur inside tokens.

(See §A for information about creating script files that contain special-valued comments that are used to generate documentation from script files.)

A *requires-comment* specifies the criteria that have to be met for its containing script to be allowed to run. The primary criterion is the version of PowerShell being used to run the script. The minimum version requirement is specified as follows:

```
#Requires -Version N[.n]
```

Where *N* is the (required) major version and *n* is the (optional) minor version.

A *requires-comment* can be present in any script file; however, it cannot be present inside a function or cmdlet. It must be the first item on a source line. A script can contain multiple *requires-comments*.

A character sequence is only recognized as a comment if that sequence begins with # or <#. For example, hello#there is considered a single token whereas hello #there is considered the token hello followed by a single-line comment. As well as following white space, the comment start sequence can also be preceded by any expression-terminating or statement-terminating character (such as), },], ', ", or ;).

A *requires-comment* cannot be present inside a snap-in.

There are four other forms of a *requires-comment*:

Syntax

```
#Requires -Assembly AssemblyId  
#Requires -Module ModuleName  
#Requires -PSSnapin PSSnapin [ -Version *N* [.n] ]  
#Requires -ShellId ShellId
```

2.2.4 White space

Syntax:

Syntax

whitespace:

Any character with Unicode class Zs, Zl, or Zp
Horizontal tab character (U+0009)
Vertical tab character (U+000B)

```
Form feed character (U+000C)
` (The backtick character U+0060) followed by new-line-character
```

Description:

White space consists of any sequence of one or more *whitespace* characters.

Except for the fact that white space may act as a separator for tokens, it is ignored.

Unlike some popular languages, PowerShell does not consider line-terminator characters ([§2.2.2](#)) to be white space. However, a line terminator can be treated as white space by preceding it immediately by a backtick character, ` (U+0060). This is necessary when the contents of a line are complete syntactically, yet the following line contains tokens intended to be associated with the previous line. For example,

PowerShell

```
$number = 10 # assigns 10 to $number; nothing is written to the pipeline
+ 20 # writes 20 to the pipeline
- 50 # writes -50 to the pipeline
$number # writes $number's value, 10, to the pipeline
```

In this example, the backtick indicates the source line is continued. The following expression is equivalent to `$number = 10 + 20 - 50`.

PowerShell

```
$number = 10 `
+ 20 `
- 50
$number # writes $number's value to the pipeline
-20
```

2.3 Tokens

Syntax:

Syntax

```
token:
  keyword
  variable
  command
  command-parameter
  command-argument-token
  integer-literal
```

```
real-literal
string-literal
type-literal
operator-or-punctuator
```

Description:

A *token* is the smallest lexical element within the PowerShell language.

Tokens can be separated by *new-lines*, comments, white space, or any combination thereof.

2.3.1 Keywords

Syntax:

Syntax

```
keyword: one of
begin      break      catch      class
continue   data       define    do
dynamicparam else      elseif    end
exit        filter     finally   for
foreach    from      function  if
in          inlinescript parallel throw
process   return     switch    param
trap       try       until    using
var        while     workflow
```

Description:

A *keyword* is a sequence of characters that has a special meaning when used in a context-dependent place. Most often, this is as the first token in a *statement*; however, there are other locations, as indicated by the grammar. (A token that looks like a keyword, but is not being used in a keyword context, is a *command-name* or a *command-argument*.)

The keywords `class`, `define`, `from`, `using`, and `var` are reserved for future use.

ⓘ Note

Editor's Note: The `class` and `using` keywords were introduced in PowerShell 5.0.

See [about Classes](#) and [about Using](#).

2.3.2 Variables

Syntax:

```
Syntax

variable:
    $$?
    $^
    $  variable-scope~opt~  variable-characters
    @  variable-scope~opt~  variable-characters
    braced-variable

braced-variable:
    ${  variable-scope~opt~  braced-variable-characters  }

variable-scope:
    Global:
    Local:
    Private:
    Script:
    Using:
    Workflow:
    variable-namespace

variable-namespace:
    variable-characters  :

variable-characters:
    variable-character
    variable-characters  variable-character

variable-character:
    A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nd
    _  (The underscore character U+005F)
    ?

braced-variable-characters:
    braced-variable-character
    braced-variable-characters  braced-variable-character

braced-variable-character:
    Any Unicode character except
        }  (The closing curly brace character U+007D)
        `  (The backtick character U+0060)
    escaped-character

escaped-character:
    `  (The backtick character U+0060) followed by any Unicode character
```

Description:

Variables are discussed in detail in §5. The variable \$? is discussed in §2.3.2.2. Scopes are discussed in §3.5.

The variables \$\$ and \$^ are reserved for use in an interactive environment, which is outside the scope of this specification.

There are two ways of writing a variable name: A *braced variable name*, which begins with \$, followed by a curly bracket-delimited set of one or more almost-arbitrary characters; and an *ordinary variable name*, which also begins with \$, followed by a set of one or more characters from a more restrictive set than a braced variable name allows. Every ordinary variable name can be expressed using a corresponding braced variable name.

PowerShell

```
$totalCost  
$Maximum_Count_26  
  
$végösszeg # Hungarian  
$и́тог # Russian  
$総計 # Japanese (Kanji)  
  
${Maximum_Count_26}  
${Name with`twhite space and `'{punctuation`'}}  
${E:\\File.txt}
```

There is no limit on the length of a variable name, all characters in a variable name are significant, and letter case is *not* distinct.

There are several different kinds of variables: user-defined (§2.3.2.1), automatic (§2.3.2.2), and preference (§2.3.2.3). They can all coexist in the same scope (§3.5).

Consider the following function definition and calls:

PowerShell

```
function Get-Power ([long]$Base, [int]$Exponent) { ... }  
  
Get-Power 5 3 # $Base is 5, $Exponent is 3  
Get-Power -Exponent 3 -Base 5 # " " "
```

Each argument is passed by position or name, one at a time. However, a set of arguments can be passed as a group with expansion into individual arguments being

handled by the runtime environment. This automatic argument expansion is known as *splatting*. For example,

```
PowerShell

$values = 5,3 # put arguments into an array
Get-Power @values

$hash = @{ Exponent = 3; Base = 5 } # put arguments into a Hashtable
Get-Power @hash

function Get-Power2 { Get-Power @args } # arguments are in an array

Get-Power2 -Exponent 3 -Base 5 # named arguments splatted named in
@args
Get-Power2 5 3 # position arguments splatted positionally in @args
```

This is achieved by using `@` instead of `$` as the first character of the variable being passed. This notation can only be used in an argument to a command.

Names are partitioned into various namespaces each of which is stored on a virtual drive ([§3.1](#)). For example, variables are stored on `Variable:`, environment variables are stored on `Env:`, functions are stored on `Function:`, and aliases are stored on `Alias:`. All of these names can be accessed as variables using the *variable-namespace* production within *variable-scope*. For example,

```
PowerShell

function F { "Hello from F" }
$Function:F # invokes function F

Set-Alias A F
$Alias:A # invokes function F via A

$Count = 10
$Variable:Count # accesses variable Count
$Env:PATH # accesses environment variable PATH
```

Any use of a variable name with an explicit `Variable:` namespace is equivalent to the use of that same variable name without that qualification. For example, `$v` and `$Variable:v` are interchangeable.

As well as being defined in the language, variables can also be defined by the cmdlet `New-Variable`.

2.3.2.1 User-defined variables

Any variable name allowed by the grammar but not used by automatic or preference variables is available for user-defined variables.

User-defined variables are created and managed by user-defined script.

2.3.2.2 Automatic variables

Automatic variables store state information about the PowerShell environment. Their values can be read in user-written script but not written.

ⓘ Note

The table originally found in this document was removed to reduce duplication. For a complete list of automatic variables, see [about Automatic Variables](#).

2.3.2.3 Preference variables

Preference variables store user preferences for the session. They are created and initialized by the PowerShell runtime environment. Their values can be read and written in user-written script.

ⓘ Note

The table originally found in this document was removed to reduce duplication. For a complete list of preference variables, see [about Preference Variables](#).

2.3.3 Commands

Syntax:

Syntax
<pre>generic-token: generic-token-parts generic-token-parts: generic-token-part generic-token-parts generic-token-part generic-token-part: expandable-string-literal verbatim-here-string-literal variable</pre>

```

generic-token-char

generic-token-char:
  Any Unicode character except
    {   }   (   )   ;   ,   |   &   $
    ` (The backtick character U+0060)
  double-quote-character
  single-quote-character
  whitespace
  new-line-character
  escaped-character

generic-token-with-subexpr-start:
  generic-token-parts $(

```

2.3.4 Parameters

Syntax:

```

Syntax

command-parameter:
  dash first-parameter-char parameter-chars colon~opt~

first-parameter-char:
  A Unicode character of classes Lu, Ll, Lt, Lm, or Lo
  _ (The underscore character U+005F)
  ?

parameter-chars:
  parameter-char
  parameter-chars parameter-char

parameter-char:
  Any Unicode character except
    { } ( ) ; , \| & .
    colon
    whitespace
    new-line-character

colon:
  : (The colon character U+003A)

verbatim-command-argument-chars:
  verbatim-command-argument-part
  verbatim-command-argument-chars verbatim-command-argument-part

verbatim-command-argument-part:
  verbatim-command-string
  & non-ampersand-character
  Any Unicode character except
    |

```

```
new-line-character

non-ampersand-character:
    Any Unicode character except &

verbatim-command-string:
    double-quote-character non-double-quote-chars
    double-quote-character

non-double-quote-chars:
    non-double-quote-char
    non-double-quote-chars non-double-quote-char

non-double-quote-char:
    Any Unicode character except
        double-quote-character
```

Description:

When a command is invoked, information may be passed to it via one or more *arguments* whose values are accessed from within the command through a set of corresponding *parameters*. The process of matching parameters to arguments is called *parameter binding*.

There are three kinds of argument:

- Switch parameter ([§8.10.5](#)) -- This has the form *command-parameter* where *first-parameter-char* and *parameter-chars* together make up the switch name, which corresponds to the name of a parameter (without its leading -) in the command being invoked. If the trailing colon is omitted, the presence of this argument indicates that the corresponding parameter be set to `$true`. If the trailing colon is present, the argument immediately following must designate a value of type `bool`, and the corresponding parameter is set to that value. For example, the following invocations are equivalent:

```
PowerShell
```

```
Set-MyProcess -Strict
Set-MyProcess -Strict: $true
```

- Parameter with argument ([§8.10.2](#)) -- This has the form *command-parameter* where *first-parameter-char* and *parameter-chars* together make up the parameter name, which corresponds to the name of a parameter (without its leading -) in the command being invoked. There must be no trailing colon. The argument immediately following designates an associated value. For example, given a

command `Get-Power`, which has parameters `$Base` and `$Exponent`, the following invocations are equivalent:

PowerShell

```
Get-Power -Base 5 -Exponent 3  
Get-Power -Exponent 3 -Base 5
```

- Positional argument ([§8.10.2](#)) - Arguments and their corresponding parameters inside commands have positions with the first having position zero. The argument in position 0 is bound to the parameter in position 0; the argument in position 1 is bound to the parameter in position 1; and so on. For example, given a command `Get-Power`, that has parameters `$Base` and `$Exponent` in positions 0 and 1, respectively, the following invokes that command:

PowerShell

```
Get-Power 5 3
```

See [§8.2](#) for details of the special parameters `--` and `--%`.

When a command is invoked, a parameter name may be abbreviated; any distinct leading part of the full name may be used, provided that is unambiguous with respect to the names of the other parameters accepted by the same command.

For information about parameter binding see [§8.14](#).

2.3.5 Literals

Syntax:

Syntax

```
literal:  
    integer-literal  
    real-literal  
    string-literal
```

2.3.5.1 Numeric literals

There are two kinds of numeric literals: integer ([§2.3.5.1.1](#)) and real ([§2.3.5.1.2](#)). Both can have multiplier suffixes ([§2.3.5.1.3](#)).

2.3.5.1.1 Integer literals

Syntax:

```
Syntax

integer-literal:
    decimal-integer-literal
    hexadecimal-integer-literal

decimal-integer-literal:
    decimal-digits numeric-type-suffix~opt~ numeric-multiplier~opt~

decimal-digits:
    decimal-digit
    decimal-digit decimal-digits

decimal-digit: one of
    0 1 2 3 4 5 6 7 8 9

numeric-type-suffix:
    long-type-suffix
    decimal-type-suffix

hexadecimal-integer-literal:
    0x hexadecimal-digits long-type-suffix~opt~
    numeric-multiplier~opt~

hexadecimal-digits:
    hexadecimal-digit
    hexadecimal-digit decimal-digits

hexadecimal-digit: one of
    0 1 2 3 4 5 6 7 8 9 a b c d e f

long-type-suffix:
    l

numeric-multiplier: one of
    kb mb gb tb pb
```

Description:

The type of an integer literal is determined by its value, the presence or absence of *long-type-suffix*, and the presence of a *numeric-multiplier* ([§2.3.5.1.3](#)).

For an integer literal with no *long-type-suffix*

- If its value can be represented by type int ([§4.2.3](#)), that is its type;
- Otherwise, if its value can be represented by type long ([§4.2.3](#)), that is its type.

- Otherwise, if its value can be represented by type decimal ([§2.3.5.1.2](#)), that is its type.
- Otherwise, it is represented by type double ([§2.3.5.1.2](#)).

For an integer literal with *long-type-suffix*

- If its value can be represented by type long ([§4.2.3](#)), that is its type;
- Otherwise, that literal is ill formed.

In the two's-complement representation of integer values, there is one more negative value than there is positive. For the int type, that extra value is -2147483648. For the long type, that extra value is -9223372036854775808. Even though the token 2147483648 would ordinarily be treated as a literal of type long, if it is preceded immediately by the unary - operator, that operator and literal are treated as a literal of type int having the smallest value. Similarly, even though the token 9223372036854775808 would ordinarily be treated as a real literal of type decimal, if it is immediately preceded by the unary - operator, that operator and literal are treated as a literal of type long having the smallest value.

Some examples of integer literals are 123 (int), 123L (long), and 200000000000 (long).

There is no such thing as an integer literal of type byte.

2.3.5.1.2 Real literals

Syntax:

```

Syntax

real-literal:
    decimal-digits . decimal-digits exponent-part~opt~ decimal-type-
suffix~opt~ numeric-multiplier~opt~
    . decimal-digits exponent-part~opt~ decimal-type-suffix~opt~ numeric-
multiplier~opt~
    decimal-digits exponent-part decimal-type-suffix~opt~ numeric-
multiplier~opt~

exponent-part:
    e sign~opt~ decimal-digits

sign: one of
    +
    dash

decimal-type-suffix:
    d
    l

```

```
numeric-multiplier: one of
  kb mb gb tb pb

dash:
  - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)
```

Description:

A real literal may contain a *numeric-multiplier* ([§2.3.5.1.3](#)).

There are two kinds of real literal: *double* and *decimal*. These are indicated by the absence or presence, respectively, of *decimal-type-suffix*. (There is no such thing as a *float real literal*.)

A double real literal has type `double` ([§4.2.4.1](#)). A decimal real literal has type `decimal` ([§4.2.4.2](#)). Trailing zeros in the fraction part of a decimal real literal are significant.

If the value of *exponent-part's decimal-digits* in a double real literal is less than the minimum supported, the value of that double real literal is 0. If the value of *exponent-part's decimal-digits* in a decimal real literal is less than the minimum supported, that literal is ill formed. If the value of *exponent-part's decimal-digits* in a double or decimal real literal is greater than the maximum supported, that literal is ill formed.

Some examples of double real literals are `1.`, `1.23`, `.45e35`, `32.e+12`, and `123.456E-231`.

Some examples of decimal real literals are `1d` (which has scale 0), `1.20d` (which has scale 2), `1.23450e1d` (i.e., `12.3450`, which has scale 4), `1.2345e3d` (i.e., `1234.5`, which has scale 1), `1.2345e-1d` (i.e., `0.12345`, which has scale 5), and `1.2345e-3d` (i.e., `0.0012345`, which has scale 7).

ⓘ Note

Because a double real literal need not have a fraction or exponent part, the grouping parentheses in `(123).M` are needed to ensure that the property or method `M` is being selected for the integer object whose value is 123. Without those parentheses, the real literal would be ill-formed.

ⓘ Note

Although PowerShell does not provide literals for infinities and NaNs, double real literal-like equivalents can be obtained from the static read-only properties

`PositiveInfinity`, `NegativeInfinity`, and `NaN` of the types `float` and `double` ([§4.2.4.1](#)).

The grammar permits what starts out as a double real literal to have an `1` or `L` type suffix. Such a token is really an integer literal whose value is represented by type long.

ⓘ Note

This feature has been retained for backwards compatibility with earlier versions of PowerShell. However, programmers are discouraged from using integer literals of this form as they can easily obscure the literal's actual value. For example, `1.2L` has value 1, `1.2345e1L` has value 12, and `1.2345e-5L` has value 0, none of which is immediately obvious.

2.3.5.1.3 Multiplier suffixes

Syntax:

Syntax

```
numeric-multiplier: *one of*
    kb mb gb tb pb
```

Description:

For convenience, integer and real literals can contain a *numeric-multiplier*, which indicates one of a set of commonly used powers of 10. *numeric-multiplier* can be written in any combination of upper- or lowercase letters.

[\[+\] Expand table](#)

Multiplier	Meaning	Example
kb	kilobyte (1024)	<code>1kb</code> \equiv 1024
mb	megabyte (1024 x 1024)	<code>1.30Dmb</code> \equiv 1363148.80
gb	gigabyte (1024 x 1024 x 1024)	<code>0x10Gb</code> \equiv 17179869184
tb	terabyte (1024 x 1024 x 1024 x 1024)	<code>1.4e23tb</code> \equiv 1.5393162788864E+35
pb	petabyte (1024 x 1024 x 1024 x 1024 x 1024)	<code>0x12Lpb</code> \equiv 20266198323167232

2.3.5.2 String literals

Syntax:

Syntax

```
string-literal:
    expandable-string-literal
    expandable-here-string-literal
    verbatim-string-literal
    verbatim-here-string-literal

expandable-string-literal:
    double-quote-character expandable-string-characters~opt~ dollars~opt~
double-quote-character

double-quote-character:
    " (U+0022)
    Left double quotation mark (U+201C)
    Right double quotation mark (U+201D)
    Double low-9 quotation mark (U+201E)

expandable-string-characters:
    expandable-string-part
    expandable-string-characters
    expandable-string-part

expandable-string-part:
    Any Unicode character except
        $
        double-quote-character
        ` (The backtick character U+0060)
    braced-variable
    $ Any Unicode character except
        (
        {
        double-quote-character
        ` (The backtick character U+0060)*
    $ escaped-character
    escaped-character
    double-quote-character double-quote-character

dollars:
    $
    dollars $

expandable-here-string-literal:
    @ double-quote-character whitespace~opt~ new-line-character
        expandable-here-string-characters~opt~ new-line-character double-
quote-character @

expandable-here-string-characters:
    expandable-here-string-part
    expandable-here-string-characters expandable-here-string-part

expandable-here-string-part:
```

```
Any Unicode character except
$  
new-line-character  
braced-variable  
$ Any Unicode character except
(  
    new-line-character  
$ new-line-character Any Unicode character except double-quote-char  
$ new-line-character double-quote-char Any Unicode character except @  
new-line-character Any Unicode character except double-quote-char  
new-line-character double-quote-char Any Unicode character except @  
  
expandable-string-with-subexpr-start:  
    double-quote-character expandable-string-chars~opt~ $()  
  
expandable-string-with-subexpr-end:  
    double-quote-char  
  
expandable-here-string-with-subexpr-start:  
    @ double-quote-character whitespace~opt~ new-line-character  
expandable-here-string-chars~opt~ $()  
  
expandable-here-string-with-subexpr-end:  
    new-line-character double-quote-character @  
  
verbatim-string-literal:  
    single-quote-character verbatim-string-characters~opt~ single-quote-char  
  
single-quote-character:  
    ' (U+0027)  
    Left single quotation mark (U+2018)  
    Right single quotation mark (U+2019)  
    Single low-9 quotation mark (U+201A)  
    Single high-reversed-9 quotation mark (U+201B)  
  
verbatim-string-characters:  
    verbatim-string-part  
    verbatim-string-characters verbatim-string-part  
  
verbatim-string-part:  
    *Any Unicode character except* single-quote-character  
    single-quote-character single-quote-character  
  
verbatim-here-string-literal:  
    @ single-quote-character whitespace~opt~ new-line-character  
        verbatim-here-string-characters~opt~ new-line-character  
        single-quote-character *@*  
  
verbatim-*here-string-characters:  
    verbatim-here-string-part  
    verbatim-here-string-characters verbatim-here-string-part  
  
verbatim-here-string-part:  
    Any Unicode character except* new-line-character  
    new-line-character Any Unicode character except single-quote-character
```

new-line-character single-quote-character Any Unicode character except

@

Description:

There are four kinds of string literals:

- *verbatim-string-literal* (single-line single-quoted), which is a sequence of zero or more characters delimited by a pair of *single-quote-characters*. Examples are " and 'red'.
- *expandable-string-literal* (single-line double-quoted), which is a sequence of zero or more characters delimited by a pair of *double-quote-characters*. Examples are "" and "red".
- *verbatim-here-string-literal* (multi-line single-quoted), which is a sequence of zero or more characters delimited by the character pairs @*single-quote-character* and *single-quote-character*@, respectively, all contained on two or more source lines. Examples are:

PowerShell

```
@'  
'@  
  
@'  
line 1  
'@  
  
@'  
line 1  
line 2  
'@
```

- *expandable-here-string-literal* (multi-line double-quoted), which is a sequence of zero or more characters delimited by the character pairs @*double-quote-character* and *double-quote-character*@, respectively, all contained on two or more source lines. Examples are:

PowerShell

```
@"  
"@  
  
@"  
line 1  
"@"
```

```
@"
line 1
line 2
"@
```

For *verbatim-here-string-literals* and *expandable-here-string-literals*, except for white space (which is ignored) no characters may follow on the same source line as the opening delimiter-character pair, and no characters may precede on the same source line as the closing delimiter character pair.

The *body* of a *verbatim-here-string-literal* or an *expandable-here-string-literal* begins at the start of the first source line following the opening delimiter, and ends at the end of the last source line preceding the closing delimiter. The body may be empty. The line terminator on the last source line preceding the closing delimiter is not part of that literal's body.

A literal of any of these kinds has type string ([§4.3.1](#)).

The character used to delimit a *verbatim-string-literal* or *expandable-string-literal* can be contained in such a string literal by writing that character twice, in succession. For example, `'What''s the time?'` and `"I said, ""Hello""."`. However, a *single-quote-character* has no special meaning inside an *expandable-string-literal*, and a *double-quote-character* has no special meaning inside a *verbatim-string-literal*.

An *expandable-string-literal* and an *expandable-here-string-literal* may contain *escaped-characters* ([§2.3.7](#)). For example, when the following string literal is written to the pipeline, the result is as shown below:

PowerShell

```
"column1`tcolumn2`nsecond line, `"Hello`", ```Q`5`!"
```

Output

```
column1<horizontal-tab>column2<new-line>
second line, "Hello", `Q5!
```

If an *expandable-string-literal* or *expandable-here-string-literal* contains the name of a variable, unless that name is preceded immediately by an escape character, it is replaced by the string representation of that variable's value ([§6.7](#)). This is known as *variable substitution*.

ⓘ Note

If the variable name is part of some larger expression, only the variable name is replaced. For example, if `$a` is an array containing the elements 100 and 200, `">$a.Length<"` results in `>100 200.Length<` while `">$($a.Length)<"` results in `>2<`. See sub-expression expansion below.

For example, the source code

PowerShell

```
$count = 10  
"The value of `$count is $count"
```

results in the *expandable-string-literal*

Output

```
The value of $count is 10.
```

Consider the following:

PowerShell

```
$a = "red", "blue"  
``$a[0] is $a[0], `$a[0] is $($a[0])" # second [0] is taken literally
```

The result is

Output

```
$a[0] is red blue[0], $a[0] is red
```

expandable-string-literals and *expandable-here-string-literals* also support a kind of substitution called *sub-expression expansion*, by treating text of the form `$(...)` as a *sub-expression* ([§7.1.6](#)). Such text is replaced by the string representation of that expression's value ([§6.8](#)). Any white space used to separate tokens within *sub-expression's statement-list* is ignored as far as the result string's construction is concerned.

The examples,

PowerShell

```
$count = 10
"$count + 5 is $($count + 5)"
"$count + 5 is `$(($count + 5))"
"$count + 5 is `$(`$count + 5)"
```

result in the following *expandable-string-literals*:

Output

```
10 + 5 is 15
10 + 5 is $(10 + 5)
10 + 5 is $($count + 5)
```

The following source,

PowerShell

```
$i = 5; $j = 10; $k = 15
``$i, `$j, and `\$k have the values $($i; $j; $k)"
```

results in the following *expandable-string-literal*:

Output

```
$i, $j, and $k have the values 5 10 15
```

These four lines could have been written more succinctly as follows:

PowerShell

```
``$i, `$j, and `\$k have the values $($i = 5); ($j = 10); ($k = 15))"
```

In the following example,

PowerShell

```
"First 10 squares: $($for ($i = 1; $i -le 10; ++$i) { "$i $($i*$i) " })"
```

the resulting *expandable-string-literal* is as follows:

Output

```
First 10 squares: 1 1 2 4 3 9 4 16 5 25 6 36 7 49 8 64 9 81 10 100
```

As shown, a *sub-expression* can contain string literals having both variable substitution and sub-expression expansion. Note also that the inner *expandable-string-literal*'s delimiters need not be escaped; the fact that they are inside a *sub-expression* means they cannot be terminators for the outer *expandable-string-literal*.

An *expandable-string-literal* or *expandable-here-string-literal* containing a variable substitution or sub-expression expansion is evaluated each time that literal is used; for example,

PowerShell

```
$a = 10
\$s1 = "`$a = $($a; ++$a)"
``\$s1 = >$s1<
\$s2 = "`$a = $($a; ++$a)"
``\$s2 = >$s2<
\$s2 = \$s1
``\$s2 = >$s2<"
```

which results in the following *expandable-string-literals*:

Output

```
$s1 = >$a = 10<
$s2 = >$a = 11<
\$s2 = >$a = 10<
```

The contents of a *verbatim-here-string-literal* are taken verbatim, including any leading or trailing white space within the body. As such, embedded *single-quote-characters* need not be doubled-up, and there is no substitution or expansion. For example,

PowerShell

```
$lit = @@
That's it!
2 * 3 = $(2*3)
'@
```

which results in the literal

Output

```
That's it!
2 * 3 = $(2*3)
```

The contents of an *expandable-here-string-literal* are subject to substitution and expansion, but any leading or trailing white space within the body but outside any *sub-expressions* is taken verbatim, and embedded *double-quote-characters* need not be doubled-up. For example,

PowerShell

```
$lit = @@
That's it!
2 * 3 = $(2*3)
"@
```

which results in the following literal when expanded:

PowerShell

```
That's it!
2 * 3 = 6
```

For both *verbatim-here-string-literals* and *expandable-here-string-literals*, each line terminator within the body is represented in the resulting literal in an implementation-defined manner. For example, in

PowerShell

```
$lit = @@
abc
xyz
"@
```

the second line of the body has two leading spaces, and the first and second lines of the body have line terminators; however, the terminator for the second line of the body is *not* part of that body. The resulting literal is equivalent to: "abc<implementation-defined character sequence>xyz".

ⓘ Note

To aid readability of source, long string literals can be broken across multiple source lines without line terminators being inserted. This is done by writing each

part as a separate literal and concatenating the parts with the + operator ([§7.7.2](#)).

This operator allows its operands to designate any of the four kinds of string literal.

ⓘ Note

Although there is no such thing as a character literal per se, the same effect can be achieved by accessing the first character in a 1-character string, as follows:

`[char]"A"` or `"A"[0]`.

For both *verbatim-here-string-literals* and *expandable-here-string-literals*, each line terminator within the body is represented exactly as it was provided.

2.3.5.3 Null literal

See the automatic variable `$null` ([§2.3.2.2](#)).

2.3.5.4 Boolean literals

See the automatic variables `$false` and `$true` ([§2.3.2.2](#)).

2.3.5.5 Array literals

PowerShell allows expressions of array type ([§9](#)) to be written using the unary comma operator ([§7.2.1](#)), *array-expression* ([§7.1.7](#)), the binary comma operator ([§7.3](#)), and the range operator ([§7.4](#)).

2.3.5.6 Hash literals

PowerShell allows expressions of type Hashtable ([§10](#)) to be written using a *hash-literal-expression* ([§7.1.9](#))

2.3.5.7 Type names

Syntax:

Syntax

```
type-name:  
    type-identifier  
    type-name . type-identifier
```

```

type-identifier:
    type-characters

type-characters:
    type-character
    type-characters type-character

type-character:
    A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nd
    _ (The underscore character U+005F)

array-type-name:
    type-name [

generic-type-name:
    type-name [

```

2.3.6 Operators and punctuators

Syntax:

```

Syntax

operator-or-punctuator: one of
    { } [ ] ( ) @( @{ $( ;
    && || & | , ++ .. :: .
    ! * / % + - --
    -and -band -bnot -bor
    -bxor -not -or -xor
    assignment-operator
    merging-redirection-operator
    file-redirection-operator
    comparison-operator
    format-operator

assignment-operator: one of
    = -= += *= /= %=

file-redirection-operator: one of
    > >> 2> 2>> 3> 3>> 4> 4>>
    5> 5>> 6> 6>> *> *>> <

merging-redirection-operator: one of
    *>&1 2>&1 3>&1 4>&1 5>&1 6>&1
    *>&2 1>&2 3>&2 4>&2 5>&2 6>&2

comparison-operator: *one of
    -as           -ccontains      -ceq
    -cge          -cgt           -cle
    -clike         -clt           -cmatch
    -cne          -cnotcontains   -cnotlike
    -cnotmatch    -contains       -creplace

```

```
-csplit      -eq          -ge
-gt          -icontains   -ieq
-ige         -igt          -ile
-ilike       -ilt          -imatch
-in          -ine          -inotcontains
-inotlike    -inotmatch  -ireplace
-is          -isnot        -isplit
-join        -le           -like
-lt          -match        -ne
-notcontains -notin        -notlike
-notmatch    -replace     -shl*
-shr         -split
```

format-operator:

```
-f
```

Description:

`&&` and `||` are reserved for future use.

① Note

Editor's Note: The pipeline chain operators `&&` and `||` were introduced in PowerShell 7. See [about Pipeline Chain Operators](#).

The name following *dash* in an operator is reserved for that purpose only in an operator context.

An operator that begins with *dash* must not have any white space between that *dash* and the token that follows it.

2.3.7 Escaped characters

Syntax:

Syntax

```
escaped-character:
` (The backtick character U+0060) followed by any Unicode character
```

Description:

An *escaped character* is a way to assign a special interpretation to a character by giving it a prefix Backtick character (U+0060). The following table shows the meaning of each *escaped-character*:

Escaped Character	Meaning
` a	Alert (U+0007)
` b	Backspace (U+0008)
` f	Form-feed (U+000C)
` n	New-line (U+000A)
` r	Carriage return (U+000D)
` t	Horizontal tab (U+0009)
` v	Vertical tab (U+0009)
` '	Single quote (U+0027)
` "	Double quote (U+0022)
` `	Backtick (U+0060)
` 0	NUL (U+0000)
` x	If x is a character other than those characters shown above, the backtick character is ignored and x is taken literally.

The implication of the final entry in the table above is that spaces that would otherwise separate tokens can be made part of a token instead. For example, a file name containing a space can be written as `Test` Data.txt` (as well as `'Test Data.txt'` or `"Test Data.txt"`).

3. Basic concepts

Article • 09/15/2023

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at: <https://www.microsoft.com/download/details.aspx?id=36389> ↗

That Word document has been converted for presentation here on Microsoft Learn.

During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

3.1 Providers and drives

A *provider* allows access to data and components that would not otherwise be easily accessible at the command line. The data is presented in a consistent format that resembles a file system drive.

The data that a provider exposes appears on a *drive*, and the data is accessed via a *path* just like with a disk drive. Built-in cmdlets for each provider manage the data on the provider drive.

PowerShell includes the following set of built-in providers to access the different types of data stores:

 Expand table

Provider	Drive Name	Description	Ref.
Alias	Alias:	PowerShell aliases	§3.1.1
Environment	Env:	Environment variables	§3.1.2
FileSystem	A; B; C; ...	Disk drives, directories, and files	§3.1.3
Function	Function:	PowerShell functions	§3.1.4

Provider	Drive Name	Description	Ref.
Variable	Variable:	PowerShell variables	§3.1.5

Windows PowerShell:

 [Expand table](#)

Provider	Drive Name	Description
Certificate	Cert:	x509 certificates for digital signatures
Registry	HKLM: (HKEY_LOCAL_MACHINE), HKCU: (HKEY_CURRENT_USER)	Windows registry
WSMan	WSMan:	WS-Management configuration information

The following cmdlets deal with providers and drives:

- [Get-PSProvider](#): Gets information about one or more providers
- [Get-PSDrive](#): Gets information about one or more drives

The type of an object that represents a provider is described in [§4.5.1](#). The type of an object that represents a drive is described in [§4.5.2](#).

3.1.1 Aliases

An *alias* is an alternate name for a command. A command can have multiple aliases, and the original name and all of its aliases can be used interchangeably. An alias can be reassigned. An alias is an item ([§3.3](#)).

An alias can be assigned to another alias; however, the new alias is not an alias of the original command.

The provider Alias is a flat namespace that contains only objects that represent the aliases. The variables have no child items.

PowerShell comes with a set of built-in aliases.

The following cmdlets deal with aliases:

- [New-Alias](#): Creates an alias
- [Set-Alias](#): Creates or changes one or more aliases
- [Get-Alias](#): Gets information about one or more aliases

- **Export-Alias:** Exports one or more aliases to a file

When an alias is created for a command using `New-Alias`, parameters to that command cannot be included in that alias. However, direct assignment to a variable in the Alias: namespace does permit parameters to be included.

(!) Note

It is a simple matter, however, to create a function that does nothing more than contain the invocation of that command with all desired parameters, and to assign an alias to that function.

The type of an object that represents an alias is described in [§4.5.4](#).

Alias objects are stored on the drive Alias: ([§3.1](#)).

3.1.2 Environment variables

The PowerShell Environment provider allows operating system environment variables to be retrieved, added, changed, cleared, and deleted.

The provider Environment is a flat namespace that contains only objects that represent the environment variables. The variables have no child items.

An environment variable's name cannot include the equal sign (=).

Changes to the environment variables affect the current session only.

An environment variable is an item ([§3.3](#)).

The type of an object that represents an environment variable is described in [§4.5.6](#).

Environment variable objects are stored on the drive Env: ([§3.1](#)).

3.1.3 File system

The PowerShell FileSystem provider allows directories and files to be created, opened, changed, and deleted.

The FileSystem provider is a hierarchical namespace that contains objects that represent the underlying file system.

Files are stored on drives with names like A:, B:, C:, and so on ([§3.1](#)). Directories and files are accessed using path notation ([§3.4](#)).

A directory or file is an item ([§3.3](#)).

3.1.4 Functions

The PowerShell Function provider allows functions ([§8.10](#)) and filters ([§8.10.1](#)) to be retrieved, added, changed, cleared, and deleted.

The provider Function is a flat namespace that contains only the function and filter objects. Neither functions nor filters have child items.

Changes to the functions affect the current session only.

A function is an item ([§3.3](#)).

The type of an object that represents a function is described in [§4.5.10](#). The type of an object that represents a filter is described in [§4.5.11](#).

Function objects are stored on drive Function: ([§3.1](#)).

3.1.5 Variables

Variables can be defined and manipulated directly in the PowerShell language.

The provider Variable is a flat namespace that contains only objects that represent the variables. The variables have no child items.

The following cmdlets also deal with variables:

- [New-Variable](#): Creates a variable
- [Set-Variable](#): Creates or changes the characteristics of one or more variables
- [Get-Variable](#): Gets information about one or more variables
- [Clear-Variable](#): Deletes the value of one or more variables
- [Remove-Variable](#): Deletes one or more variables

As a variable is an item ([§3.3](#)), it can be manipulated by most Item-related cmdlets.

The type of an object that represents a variable is described in [§4.5.3](#).

Variable objects are stored on drive Variable: ([§3.1](#)).

3.2 Working locations

The *current working location* is the default location to which commands point. This is the location used if an explicit path ([§3.4](#)) is not supplied when a command is invoked. This location

includes the *current drive*.

A PowerShell host may have multiple drives, in which case, each drive has its own current location.

When a drive name is specified without a directory, the current location for that drive is implied.

The current working location can be saved on a stack, and then set to a new location. Later, that saved location can be restored from that stack and made the current working location. There are two kinds of location stacks: the *default working location stack*, and zero or more user-defined *named working location stacks*. When a session begins, the default working location stack is also the *current working location stack*. However, any named working location stack can be made the current working location stack.

The following cmdlets deal with locations:

- [Set-Location](#): Establishes the current working location
- [Get-Location](#): Determines the current working location for the specified drive(s), or the working locations for the specified stack(s)
- [Push-Location](#): Saves the current working location on the top of a specified stack of locations
- [Pop-Location](#): Restores the current working location from the top of a specified stack of locations

The object types that represents a working location and a stack of working locations are described in [§4.5.5](#).

3.3 Items

An *item* is an alias ([§3.1.1](#)), a variable ([§3.1.5](#)), a function ([§3.1.4](#)), an environment variable ([§3.1.2](#)), or a file or directory in a file system ([§3.1.3](#)).

The following cmdlets deal with items:

- [New-Item](#): Creates a new item
- [Set-Item](#): Changes the value of one or more items
- [Get-Item](#): Gets the items at the specified location
- [Get-ChildItem](#): Gets the items and child items at the specified location
- [Copy-Item](#): Copies one or more items from one location to another
- [Move-Item](#): Moves one or more items from one location to another
- [Rename-Item](#): Renames an item
- [Invoke-Item](#): Performs the default action on one or more items

- [Clear-Item](#): Deletes the contents of one or more items, but does not delete the items (see [§4.5.17](#))
- [Remove-Item](#): Deletes the specified items

The following cmdlets deal with the content of items:

- [Get-Content](#): Gets the content of the item
- [Add-Content](#): Adds content to the specified items
- [Set-Content](#): Writes or replaces the content in an item
- [Clear-Content](#): Deletes the contents of an item

The type of an object that represents a directory is described in [§4.5.17](#). The type of an object that represents a file is described in [§4.5.18](#).

3.4 Path names

All items in a data store accessible through a PowerShell provider can be identified uniquely by their path names. A *path name* is a combination of the item name, the container and subcontainers in which the item is located, and the PowerShell drive through which the containers are accessed.

Path names are divided into one of two types: fully qualified and relative. A *fully qualified path name* consists of all elements that make up a path. The following syntax shows the elements in a fully qualified path name:

Tip

The `~opt~` notation in the syntax definitions indicates that the lexical entity is optional in the syntax.

Syntax

```
path:  
    provider~opt~  drive~opt~  containers~opt~  item  
  
provider:  
    module~opt~  provider  ::  
  
module:  
    module-name  \  
  
drive:  
    drive-name  :  
  
containers:
```

```
container  \
  containers container  \
```

module-name refers to the parent module.

provider refers to the PowerShell provider through which the data store is accessed.

drive refers to the PowerShell drive that is supported by a particular PowerShell provider.

A *container* can contain other containers, which can contain other containers, and so on, with the final container holding an *item*. Containers must be specified in the hierarchical order in which they exist in the data store.

Here is an example of a path name:

```
E:\Accounting\InvoiceSystem\Production\MasterAccount\MasterFile.dat
```

If the final element in a path contains other elements, it is a *container element*; otherwise, it's a *leaf element*.

In some cases, a fully qualified path name is not needed; a relative path name will suffice. A *relative path name* is based on the current working location. PowerShell allows an item to be identified based on its location relative to the current working location. A relative path name involves the use of some special characters. The following table describes each of these characters and provides examples of relative path names and fully qualified path names. The examples in the table are based on the current working directory being set to C:\Windows:

 Expand table

Symbol	Description	Relative path	Fully qualified path
.	Current working location	.\System	C:\Windows\System
..	Parent of the current working location	..\Program Files	C:\Program Files
\	Drive root of the current working location	\Program Files	C:\Program Files
none	No special characters	System	C:\Windows\System

To use a path name in a command, enter that name as a fully qualified or relative path name.

The following cmdlets deal with paths:

- [Convert-Path](#): Converts a path from a PowerShell path to a PowerShell provider path
- [Join-Path](#): Combines a path and a child path into a single path
- [Resolve-Path](#): Resolves the wildcard characters in a path

- [Split-Path](#): Returns the specified part of a path
- [Test-Path](#): Determines whether the elements of a path exist or if a path is well formed

Some cmdlets (such as [Add-Content](#) and [Copy-Item](#)) use file filters. A *file filter* is a mechanism for specifying the criteria for selecting from a set of paths.

The object type that represents a resolved path is described in [§4.5.5](#). Paths are often manipulated as strings.

3.5 Scopes

3.5.1 Introduction

A name can denote a variable, a function, an alias, an environment variable, or a drive. The same name may denote different items at different places in a script. For each different item that a name denotes, that name is visible only within the region of script text called its *scope*. Different items denoted by the same name either have different scopes, or are in different name spaces.

Scopes may nest, in which case, an outer scope is referred to as a *parent scope*, and any nested scopes are *child scopes* of that parent. The scope of a name is the scope in which it is defined and all child scopes, unless it is made private. Within a child scope, a name defined there hides any items defined with the same name in parent scopes.

Unless dot source notation ([§3.5.5](#)) is used, each of the following creates a new scope:

- A script file
- A script block
- A function or filter

Consider the following example:

```
PowerShell

# Start of script
$x = 2; $y = 3
Get-Power $x $y

# Function defined in script
function Get-Power([int]$x, [int]$y) {
    if ($y -gt 0) {
        return $x * (Get-Power $x (--$y))
    } else {
        return 1
    }
}
```

```
}
```

```
# End of script
```

The scope of the variables `$x` and `$y` created in the script is the body of that script, including the function defined inside it. Function `Get-Power` defines two parameters with those same names. As each function has its own scope, these variables are different from those defined in the parent scope, and they hide those from the parent scope. The function scope is nested inside the script scope.

Note that the function calls itself recursively. Each time it does so, it creates yet another nested scope, each with its own variables `$x` and `$y`.

Here is a more complex example, which also shows nested scopes and reuse of names:

```
PowerShell
```

```
# start of script scope
$x = 2          # top-level script-scope $x created
# $x is 2
F1             # create nested scope with call to function F1
# $x is 2
F3             # create nested scope with call to function F3
# $x is 2

function F1 {      # start of function scope
# $x is 2
    $x = $true     # function-scope $x created
# $x is $true

    & {           # create nested scope with script block
# $x is $true
        $x = 12.345 # scriptblock-scope $x created
# $x is 12.345
    }             # end of scriptblock scope, local $x goes away

# $x is $true
F2             # create nested scope with call to function F2
# $x is $true
}               # end of function scope, local $x goes away

function F2 {      # start of function scope
# $x is $true
    $x = "red"     # function-scope $x created
# $x is "red"
}               # end of function scope, local $x goes away

function F3 {      # start of function scope
# $x is 2
    if ($x -gt 0) {
# $x is 2
        $x = "green"
```

```
        # $x is "green"
    }
    # end of block, but not end of any scope
    # $x is still "green"
}
# end of function scope, local $x goes away
# end of script scope
```

3.5.2 Scope names and numbers

PowerShell supports the following scopes:

- Global: This is the top-most level scope. All automatic and preference variables are defined in this scope. The global scope is the parent scope of all other scopes, and all other scopes are child scopes of the global scope.
- Local: This is the current scope at any execution point within a script, script block, or function. Any scope can be the local scope.
- Script: This scope exists for each script file that is executed. The script scope is the parent scope of all scopes created from within it. A script block does *not* have its own script scope; instead, its script scope is that of its nearest ancestor script file. Although there is no such thing as module scope, script scope provides the equivalent.

Names can be declared private, in which case, they are not visible outside of their parent scope, not even to child scopes. The concept of private is not a separate scope; it's an alias for local scope with the addition of hiding the name if used as a writable location.

Scopes can be referred to by a number, which describes the relative position of one scope to another. Scope 0 denotes the local scope, scope 1 denotes a 1-generation ancestor scope, scope 2 denotes a 2-generation ancestor scope, and so on. (Scope numbers are used by cmdlets that manipulate variables.)

3.5.3 Variable name scope

As shown by the following production, a variable name can be specified with any one of six different scopes:

Syntax

```
variable-scope:
  Global:
  Local:
  Private:
  Script:
  Using:
```

Workflow: variable-namespace

The scope is optional. The following table shows the meaning of each in all possible contexts. It also shows the scope when no scope is specified explicitly:

[Expand table](#)

Scope Modifier	Within a Script File	Within a Script Block	Within a Function
Global	Global scope	Global scope	Global scope
Script	Nearest ancestor script file's scope or Global if there is no nearest ancestor script file	Nearest ancestor script file's scope or Global if there is no nearest ancestor script file	Nearest ancestor script file's scope or Global if there is no nearest ancestor script file
Private	Global/Script/Local scope	Local scope	Local scope
Local	Global/Script/Local scope	Local scope	Local scope
Using	Implementation defined	Implementation defined	Implementation defined
Workflow	Implementation defined	Implementation defined	Implementation defined
None	Global/Script/Local scope	Local scope	Local scope

Variable scope information can also be specified when using the family of cmdlets listed in [\(\\$3.1.5\)](#). In particular, refer to the parameter `Scope`, and the parameters `Option Private` and `Option AllScope` for more information.

The `Using:` scope modifier is used to access variables defined in another scope while running scripts via cmdlets like `Start-Job`, `Invoke-Command`, or within an *inlinescript-statement*. For example:

PowerShell

```
$a = 42
Invoke-Command --ComputerName RemoteServer { $Using:a } # returns 42
workflow foo
{
    $b = "Hello"
    inlinescript { $Using:b }
}
foo # returns "Hello"
```

The scope workflow is used with a *parallel-statement* or *sequence-statement* to access a variable defined in the workflow.

3.5.4 Function name scope

A function name may also have one of the four different scopes, and the visibility of that name is the same as for variables ([§3.5.3](#)).

3.5.5 Dot source notation

When a script file, script block, or function is executed from within another script file, script block, or function, the executed script file creates a new nested scope. For example,

```
PowerShell  
  
Script1.ps1  
& "Script1.ps1"  
& { ... }  
FunctionA
```

However, when *dot source notation* is used, no new scope is created before the command is executed, so additions/changes it would have made to its own local scope are made to the current scope instead. For example,

```
PowerShell  
  
. Script2.ps1  
. "Script2.ps1"  
. { ... }  
. FunctionA
```

3.5.6 Modules

Just like a top-level script file is at the root of a hierarchical nested scope tree, so too is each module ([§3.14](#)). However, by default, only those names exported by a module are available by name from within the importing context. The Global parameter of the cmdlet [Import-Module](#) allows exported names to have increased visibility.

3.6 ReadOnly and Constant Properties

Variables and aliases are described by objects that contain a number of properties. These properties are set and manipulated by two families of cmdlets ([§3.1.5](#), [§3.1.1](#)). One such

property is Options, which can be set to ReadOnly or Constant (using the Option parameter). A variable or alias marked ReadOnly can be removed, and its properties can be changed provided the Force parameter is specified. However, a variable or alias marked Constant cannot be removed nor have its properties changed.

3.7 Method overloads and call resolution

3.7.1 Introduction

As stated in §1, an external procedure made available by the execution environment (and written in some language other than PowerShell) is called a *method*.

The name of a method along with the number and types of its parameters are collectively called that method's *signature*. (Note that the signature does not include the method's return type.) The execution environment may allow a type to have multiple methods with the same name provided each has a different signature. When multiple versions of some method are defined, that method is said to be *overloaded*. For example, the type Math ([§4.3.8](#)) contains a set of methods called `Abs`, which computes the absolute value of a specified number, where the specified number can have one of a number of types. The methods in that set have the following signatures:

```
PowerShell

Abs(decimal)
Abs(float)
Abs(double)
Abs(int)
Abs(long)
Abs(SByte)
Abs(Int16)
```

In this case, all of the methods have the same number of arguments; their signatures differ by argument type only.

Another example involves the type Array ([§4.3.2](#)), which contains a set of methods called Copy that copies a range of elements from one array to another, starting at the beginning of each array (by default) or at some designated element. The methods in that set have the following signatures:

```
PowerShell

Copy(Array, Array, int)
Copy(Array, Array, long)
```

```
Copy(Array, int, Array, int, int)
Copy(Array, long, Array, long, long)
```

In this case, the signatures differ by argument type and, in some cases, by argument number as well.

In most calls to overloaded methods, the number and type of the arguments passed exactly match one of the overloads, and the method selected is obvious. However, if that is not the case, there needs to be a way to resolve which overloaded version to call, if any. For example,

PowerShell

```
[Math]::Abs([byte]10) # no overload takes type byte
[array]::Copy($source, 3, $dest, 5L, 4) # both int and long indexes
```

Other examples include the type **string** (i.e.; `System.String`), which has numerous overloaded methods.

Although PowerShell has rules for resolving method calls that do not match an overloaded signature exactly, PowerShell does not itself provide a way to define overloaded methods.

! Note

Editor's Note: PowerShell 5.0 added the ability to define script-based classes. These classes can contain overloaded methods.

3.7.2 Method overload resolution

Given a method call ([§7.1.3](#)) having a list of argument expressions, and a set of *candidate methods* (i.e., those methods that could be called), the mechanism for selecting the *best method* is called *overload resolution*.

Given the set of applicable candidate methods ([§3.7.3](#)), the best method in that set is selected. If the set contains only one method, then that method is the best method. Otherwise, the best method is the one method that is better than all other methods with respect to the given argument list using the rules shown in [§3.7.4](#). If there is not exactly one method that is better than all other methods, then the method invocation is ambiguous and an error is reported.

The best method must be accessible in the context in which it is called. For example, a PowerShell script cannot call a method that is private or protected.

The best method for a call to a static method must be a static method, and the best method for a call to an instance method must be an instance method.

3.7.3 Applicable method

A method is said to be *applicable* with respect to an argument list A when one of the following is true:

- The number of arguments in A is identical to the number of parameters that the method accepts.
- The method has M required parameters and N optional parameters, and the number of arguments in A is greater than or equal to M, but less than N.
- The method accepts a variable number of arguments and the number of arguments in A is greater than the number of parameters that the method accepts.

In addition to having an appropriate number of arguments, each argument in A must match the parameter-passing mode of the argument, and the argument type must match the parameter type, or there must be a conversion from the argument type to the parameter type.

If the argument type is `ref` ([§4.3.6](#)), the corresponding parameter must also be `ref`, and the argument type for conversion purposes is the type of the property `Value` from the `ref` argument.

If the argument type is `ref`, the corresponding parameter could be `out` instead of `ref`.

If the method accepts a variable number of arguments, the method may be applicable in either *normal form* or *expanded form*. If the number of arguments in A is identical to the number of parameters that the method accepts and the last parameter is an array, then the form depends on the rank of one of two possible conversions:

- The rank of the conversion from the type of the last argument in A to the array type for the last parameter.
- The rank of the conversion from the type of the last argument in A to the element type of the array type for the last parameter.

If the first conversion (to the array type) is better than the second conversion (to the element type of the array), then the method is applicable in normal form, otherwise it is applicable in expanded form.

If there are more arguments than parameters, the method may be applicable in expanded form only. To be applicable in expanded form, the last parameter must have array type. The method is replaced with an equivalent method that has the last parameter replaced with sufficient parameters to account for each unmatched argument in A. Each additional parameter type is the element type of the array type for the last parameter in the original method. The above rules for an applicable method are applied to this new method and argument list A.

3.7.4 Better method

Given an argument list A with a set of argument expressions $\{ E_1, E_2, \dots, E_N \}$ and two application methods $M_{P\sim}$ and $M_{Q\sim}$ with parameter types $\{ P_1, P_2, \dots, P_N \}$ and $\{ Q_1, Q_2, \dots, Q_N \}$, $M_{P\sim}$ is defined to be a better method than $M_{Q\sim}$ if the *cumulative ranking of conversions* for $M_{P\sim}$ is better than that for $M_{Q\sim}$.

The cumulative ranking of conversions is calculated as follows. Each conversion is worth a different value depending on the number of parameters, with the conversion of E_1 worth N, E_2 worth N-1, down to E_N worth 1. If the conversion from $E_{X\sim}$ to $P_{X\sim}$ is better than that from $E_{X\sim}$ to $Q_{X\sim}$, the $M_{P\sim}$ accumulates $N-X+1$; otherwise, $M_{Q\sim}$ accumulates $N-X+1$. If $M_{P\sim}$ and $M_{Q\sim}$ have the same value, then the following tie breaking rules are used, applied in order:

- The cumulative ranking of conversions between parameter types (ignoring argument types) is computed in a manner similar to the previous ranking, so P_1 is compared against Q_1 , P_2 against Q_2 , ..., and P_N against Q_N . The comparison is skipped if the argument was `$null`, or if the parameter types are not numeric types. The comparison is also skipped if the argument conversion $E_{X\sim}$ loses information when converted to $P_{X\sim}$ but does not lose information when converted to $Q_{X\sim}$, or vice versa. If the parameter conversion types are compared, then if the conversion from $P_{X\sim}$ to $Q_{X\sim}$ is better than that from $Q_{X\sim}$ to $P_{X\sim}$, the $M_{P\sim}$ accumulates $N-X+1$; otherwise, $M_{Q\sim}$ accumulates $N-X+1$. This tie breaking rule is intended to prefer the *most specific method* (i.e., the method with parameters having the smallest data types) if no information is lost in conversions, or to prefer the *most general method* (i.e., the method with the parameters with the largest data types) if conversions result in loss of information.
- If both methods use their expanded form, the method with more parameters is the better method.
- If one method uses the expanded form and the other uses normal form, the method using normal form is the better method.

3.7.5 Better conversion

The text below marked like this is specific to Windows PowerShell.

Conversions are ranked in the following manner, from lowest to highest:

- $T_1[]$ to $T_2[]$ where no assignable conversion between T_1 and T_2 exists
- T to string where T is any type
- T_1 to T_2 where T_1 or T_2 define a custom conversion in an implementation-defined manner
- T_1 to T_2 where T_1 implements `IConvertible`

- `T~1~` to `T~2~` where `T~1~` or `T~2~` implements the method `T~2~ op_Implicit(T1)`
- `T~1~` to `T~2~` where `T~1~` or `T~2~` implements the method `T~2~ op_Explicit(T1)`
- `T~1~` to `T~2~` where `T~2~` implements a constructor taking a single argument of type `T~1~`
- Either of the following conversions:
 - string to `T` where `T` implements a static method `T Parse(string)` or `T Parse(string, IFormatProvider)`
 - `T~1~` to `T~2~` where `T~2~` is any enum and `T~1~` is either string or a collection of objects that can be converted to string
- `T` to `PSObject` where `T` is any type
- Any of the following conversions: `Language`
 - `T` to `bool` where `T` is any numeric type
 - string to `T` where `T` is `regex`, `wmisearcher`, `wmi`, `wmiclass`, `adsi`, `adsisearcher`, or `type`
 - `T` to `bool`
 - `T~1~` to `Nullable[T~2~]` where a conversion from `T~1~` to `T~2~` exists
 - `T` to `void`
 - `T~1~[]` to `T~2~[]` where an assignable conversion between `T~1~` and `T~2~` exists
 - `T~1~` to `T~2~[]` where `T~1~` is a collection
 - `IDictionary` to `Hashtable`
 - `T` to `ref`
 - `T` to `xml`
 - `scriptblock` to `delegate`
 - `T~1~` to `T~2~` where `T~1~` is an integer type and `T~2~` is an enum
- `$null` to `T` where `T` is any value type
- `$null` to `T` where `T` is any reference type
- Any of the following conversions:
 - byte to `T` where `T` is `SByte`
 - `UInt16` to `T` where `T` is `SByte`, `byte`, or `Int16`
 - `Int16` to `T` where `T` is `SByte` or `byte`
 - `UInt32` to `T` where `T` is `SByte`, `byte`, `Int16`, `UInt16`, or `int`
 - `int` to `T` where `T` is `SByte`, `byte`, `Int16`, or `UInt16`
 - `UInt64` to `T` where `T` is `SByte`, `byte`, `Int16`, `UInt16`, `int`, `UInt32`, or `long`
 - `long` to `T` where `T` is `SByte`, `byte`, `Int16`, `UInt16`, `int`, or `UInt32`

- `float` to `T` where `T` is any integer type or `decimal`
- `double` to `T` where `T` is any integer type or `decimal`
- `decimal` to `T` where `T` is any integer type
- Any of the following conversions:
 - `SByte` to `T` where `T` is `byte`, `uint6`, `UInt32`, or `UInt64`
 - `Int16` to `T` where `T` is `UInt16`, `UInt32`, or `UInt64`
 - `int` to `T` where `T` is `UInt32` or `UInt64`
 - `long` to `UInt64`
 - `decimal` to `T` where `T` is `float` or `double`
- Any of the following conversions:
 - `T` to `string` where `T` is any numeric type
 - `T` to `char` where `T` is any numeric type
 - `string` to `T` where `T` is any numeric type
- Any of the following conversions, these conversion are considered an assignable conversions:
 - `byte` to `T` where `T` is `Int16`, `UInt16`, `int`, `UInt32`, `long`, `UInt64`, `single`, `double`, or `decimal`
 - `SByte` to `T` where `T` is `Int16`, `UInt16`, `int`, `UInt32`, `long`, `UInt64`, `single`, `double`, or `decimal`
 - `UInt16` to `T` where `T` is `int`, `UInt32`, `long`, or `UInt64`, `single`, `double`, or `decimal`
 - `Int16` to `T` where `T` is `int`, `UInt32`, `long`, or `UInt64`, `single`, `double`, or `decimal`
 - `UInt32` to `T` where `T` is `long`, or `UInt64`, `single`, `double`, or `decimal`
 - `int` to `T` where `T` is `long`, `UInt64`, `single`, `double`, or `decimal`
 - `single` to `double`
- `T~1~` to `T~2~` where `T~2~` is a base class or interface of `T~1~`. This conversion is considered an assignable conversion.
- `string` to `char[]`
- `T` to `T` -- This conversion is considered an assignable conversion.

For each conversion of the form `T~1~` to `T~2~[]` where `T~1~` is not an array and no other conversion applies, if there is a conversion from `T~1~` to `T~2~`, the rank of the conversion is worse than the conversion from `T~1~` to `T~2~`, but better than any conversion ranked less than the conversion from `T~1~` to `T~2~`.

3.8 Name lookup

It is possible to have commands of different kinds all having the same name. The order in which name lookup is performed in such a case is alias, function, cmdlet, and external command.

3.9 Type name lookup

§7.1.10 contains the statement, "A *type-literal* is represented in an implementation by some unspecified *underlying type*. As a result, a type name is a synonym for its underlying type." Example of types are `int`, `double`, `long[]`, and `Hashtable`.

Type names are matched as follows: Compare a given type name with the list of built-in *type accelerators*, such as `int`, `long`, `double`. If a match is found, that is the type. Otherwise, presume the type name is fully qualified and see if such a type exists on the host system. If a match is found, that is the type. Otherwise, add the namespace prefix `System..`. If a match is found, that is the type. Otherwise, the type name is in error. This algorithm is applied for each type argument for generic types. However, there is no need to specify the arity (the number of arguments or operands taken by a function or operator).

3.10 Automatic memory management

Various operators and cmdlets result in the allocation of memory for reference-type objects, such as strings and arrays. The allocation and freeing of this memory is managed by the PowerShell runtime system. That is, PowerShell provides automatic *garbage collection*.

3.11 Execution order

A *side effect* is a change in the state of a command's execution environment. A change to the value of a variable (via the assignment operators or the pre- and post-increment and decrement operators) is a side effect, as is a change to the contents of a file.

Unless specified otherwise, statements are executed in lexical order.

Except as specified for some operators, the order of evaluation of terms in an expression and the order in which side effects take place are both unspecified.

An expression that invokes a command involves the expression that designates the command, and zero or more expressions that designate the arguments whose values are to be passed to that command. The order in which these expressions are evaluated relative to each other is unspecified.

3.12 Error handling

When a command fails, this is considered an *error*, and information about that error is recorded in an *error record*, whose type is unspecified ([§4.5.15](#)); however, this type supports subscripting.

An error falls into one of two categories. Either it terminates the operation (a *terminating error*) or it doesn't (a *non-terminating error*). With a terminating error, the error is recorded and the operation stops. With a non-terminating error, the error is recorded and the operation continues.

Non-terminating errors are written to the error stream. Although that information can be redirected to a file, the error objects are first converted to strings and important information in those objects would not be captured making diagnosis difficult if not impossible. Instead, the error text can be redirected ([§7.12](#)) and the error object saved in a variable, as in `$Error1 = command 2>&1`.

The automatic variable `$Error` contains a collection of error records that represent recent errors, and the most recent error is in `$Error[0]`. This collection is maintained in a buffer such that old records are discarded as new ones are added. The automatic variable `$MaximumErrorCount` controls the number of records that can be stored.

`$Error` contains all of the errors from all commands mixed in together in one collection. To collect the errors from a specific command, use the common parameter `ErrorVariable`, which allows a user-defined variable to be specified to hold the collection.

3.13 Pipelines

A *pipeline* is a series of one or more commands each separated by the pipe operator `|` (U+007C). Each command receives input from its predecessor and writes output to its successor. Unless the output at the end of the pipeline is discarded or redirected to a file, it is sent to the host environment, which may choose to write it to standard output. Commands in a pipeline may also receive input from arguments. For example, consider the following use of commands `Get-ChildItem`, `Sort-Object`, and `Process-File`, which create a list of file names in a given file system directory, sort a set of text records, and perform some processing on a text record, respectively:

PowerShell

```
Get-ChildItem  
Get-ChildItem E:*.txt | Sort-Object -CaseSensitive | Process-File >results.txt
```

In the first case, `Get-ChildItem` creates a collection of names of the files in the current/default directory. That collection is sent to the host environment, which, by default, writes each element's value to standard output.

In the second case, `Get-ChildItem` creates a collection of names of the files in the directory specified, using the argument `E:*.txt`. That collection is written to the command `Sort-Object`, which, by default, sorts them in ascending order, sensitive to case (by virtue of the `CaseSensitive` argument). The resulting collection is then written to command `Process-File`, which performs some (unknown) processing. The output from that command is then redirected to the file `results.txt`.

If a command writes a single object, its successor receives that object and then terminates after writing its own object(s) to its successor. If, however, a command writes multiple objects, they are delivered one at a time to the successor command, which executes once per object. This behavior is called *streaming*. In stream processing, objects are written along the pipeline as soon as they become available, not when the entire collection has been produced.

When processing a collection, a command can be written such that it can do special processing before the initial element and after the final element.

3.14 Modules

A *module* is a self-contained reusable unit that allows PowerShell code to be partitioned, organized, and abstracted. A module can contain commands (such as cmdlets and functions) and items (such as variables and aliases) that can be used as a single unit.

Once a module has been created, it must be *imported* into a session before the commands and items within it can be used. Once imported, commands and items behave as if they were defined locally. A module is imported explicitly with the `Import-Module` command. A module may also be imported automatically as determined in an implementation defined manner.

The type of an object that represents a module is described in §4.5.12.

Modules are discussed in detail in [§11](#).

3.15 Wildcard expressions

A wildcard expression may contain zero or more of the following elements:

Element	Description
Character other than *, ?, or [Matches that one character
*	Matches zero or more characters. To match a * character, use [*].
?	Matches any one character. To match a ? character, use [?].
[set]	<p>Matches any one character from <i>set</i>, which cannot be empty.</p> <p>If <i>set</i> begins with], that right square bracket is considered part of <i>set</i> and the next right square bracket terminates the set; otherwise, the first right square bracket terminates the set.</p> <p>If <i>set</i> begins or ends with -, that hyphen-minus is considered part of <i>set</i>; otherwise, it indicates a range of consecutive Unicode code points with the characters either side of the hyphen-minus being the inclusive range delimiters. For example, A-Z indicates the 26 uppercase English letters, and 0-9 indicates the 10 decimal digits.</p>

ⓘ Note

More information can be found in, [The Open Group Base Specifications: Pattern Matching](#), IEEE Std 1003.1, 2004 Edition. ⓘ . However, in PowerShell, the escape character is backtick, not backslash.

3.16 Regular expressions

A regular expression may contain zero or more of the following elements:

[] [Expand table](#)

Element	Description
Character other than ., [, ^, *, \$, or \	Matches that one character
.	Matches any one character. To match a . character, use \..
[set] [^set]	<p>The [set] form matches any one character from <i>set</i>. The [^set] form matches no characters from <i>set</i>. <i>set</i> cannot be empty.</p> <p>If <i>set</i> begins with] or ^], that right square bracket is considered part of <i>set</i> and the next right square bracket terminates the set; otherwise, the first right square bracket terminates the set.</p>

Element	Description
	If set begins with - or ^-, or ends with -, that hyphen-minus is considered part of <i>set</i> ; otherwise, it indicates a range of consecutive Unicode code points with the characters either side of the hyphen-minus being the inclusive range delimiters. For example, A-Z indicates the 26 uppercase English letters, and 0-9 indicates the 10 decimal digits.
*	Matches zero or more occurrences of the preceding element.
+	Matches one or more occurrences of the preceding element.
?	Matches zero or one occurrences of the preceding element.
^	Matches at the start of the string. To match a ^ character, use \^.
\$	Matches at the end of the string. To match a \$ character, use \\$.
\c	Escapes character <i>c</i> , so it isn't recognized as a regular expression element.

(!) Note

More information can be found in, [The Open Group Base Specifications: Regular Expressions, IEEE Std 1003.1, 2004 Edition](#).

Windows PowerShell: Character classes available in Microsoft .NET Framework regular expressions are supported, as follows:

[Expand table](#)

Element	Description
\p{name}	Matches any character in the named character class specified by <i>name</i> . Supported names are Unicode groups and block ranges such as Ll, Nd, Z, IsGreek, and IsBoxDrawing.
\P{name}	Matches text not included in the groups and block ranges specified in <i>name</i> .
\w	Matches any word character. Equivalent to the Unicode character categories [\p{L1}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}]. If ECMAScript-compliant behavior is specified with the ECMAScript option, \w is equivalent to [a-zA-Z_0-9].
\W	Matches any non-word character. Equivalent to the Unicode categories [\^\p{L1}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}].
\s	Matches any white space character. Equivalent to the Unicode character categories [\f\n\r\t\v\x85\p{Z}].
\S	Matches any non-white-space character. Equivalent to the Unicode character categories [\^\f\n\r\t\v\x85\p{Z}].

Element	Description
\d	Matches any decimal digit. Equivalent to \p{Nd} for Unicode and [0-9] for non-Unicode behavior.
\D	Matches any non-digit. Equivalent to \P{Nd} for Unicode and [\^0-9] for non-Unicode behavior.

Quantifiers available in Microsoft .NET Framework regular expressions are supported, as follows:

[Expand table](#)

Element	Description
*	Specifies zero or more matches; for example, \w* or (abc)*. Equivalent to {0,}.
+	Matches repeating instances of the preceding characters.
?	Specifies zero or one matches; for example, \w? or (abc)? . Equivalent to {0,1}.
{n}	Specifies exactly <i>n</i> matches; for example, (pizza){2}.
{n,}	Specifies at least <i>n</i> matches; for example, (abc){2,}.
{n,m}	Specifies at least <i>n</i> , but no more than <i>m</i> , matches.

4. Types

Article • 09/15/2023

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

In PowerShell, each value has a type, and types fall into one of two main categories: **value types** and **reference types**. Consider the type `int`, which is typical of value types. A value of type `int` is completely self-contained; all the bits needed to represent that value are stored in that value, and every bit pattern in that value represents a valid value for its type. Now, consider the array type `int[]`, which is typical of reference types. A so-called value of an array type can hold either a reference to an object that actually contains the array elements, or the **null reference** whose value is `$null`. The important distinction between the two type categories is best demonstrated by the differences in their semantics during assignment. For example,

PowerShell

```
$i = 100 # $i designates an int value 100
$j = $i # $j designates an int value 100, which is a copy

$a = 10,20,30 # $a designates an object[], Length 3, value 10,20,30
$b = $a # $b designates exactly the same array as does $a, not a copy
$a[1] = 50 # element 1 (which has a value type) is changed from 20 to 50
$b[1] # $b refers to the same array as $a, so $b[1] is 50
```

As we can see, the assignment of a reference type value involves a **shallow copy**; that is, a copy of the reference to the object rather than its actual value. In contrast, a **deep copy** requires making a copy of the object as well.

A **numeric type** is one that allows representation of integer or fractional values, and that supports arithmetic operations on those values. The set of numerical types includes the integer ([§4.2.3](#)) and real number ([§4.2.4](#)) types, but does not include bool ([§4.2.1](#)) or char ([§4.2.2](#)). An implementation may provide other numeric types (such as signed byte, unsigned integer, and integers of other sizes).

A **collection** is a group of one or more related items, which need not have the same type. Examples of collection types are arrays, stacks, queues, lists, and hash tables. A program can *enumerate* (or *iterate*) over the elements in a collection, getting access to each element one at a time. Common ways to do this are with the `foreach` statement ([§8.4.4](#)) and the [ForEach-Object](#) cmdlet. The type of an object that represents an enumerator is described in [§4.5.16](#).

In this chapter, there are tables that list the accessible members for a given type. For methods, the **Type** is written with the following form: *returnType/argumentTypeList*. If the argument type list is too long to fit in that column, it is shown in the **Purpose** column instead.

Other integer types are `SByte`, `Int16`, `UInt16`, `UInt32`, and `UInt64`, all in the namespace **System**.

Many collection classes are defined as part of the **System.Collections** or **System.Collections.Generic** namespaces. Most collection classes implement the interfaces `ICollection`, `IComparer`, `IEnumerable`, `IList`, `IDictionary`, and `IDictionaryEnumerator` and their generic equivalents.

You can also use shorthand names for some types. For more information, see [about_Type_Accelerators](#).

4.1 Special types

4.1.1 The void type

This type cannot be instantiated. It provides a means to discard a value explicitly using the cast operator ([§7.2.9](#)).

4.1.2 The null type

The *null* type has one instance, the automatic variable `$null` ([§2.3.2.2](#)), also known as the null value. This value provides a means for expressing "nothingness" in reference contexts. The characteristics of this type are unspecified.

4.1.3 The object type

Every type in PowerShell except the null type ([§4.1.2](#)) is derived directly or indirectly from the type object, so `object` is the ultimate base type of all non-null types. A variable constrained ([§5.3](#)) to `object` is really not constrained at all, as it can contain a value of any type.

4.2 Value types

4.2.1 Boolean

The Boolean type is `bool`. There are only two values of this type, `False` and `True`, represented by the automatic variables `$false` and `$true`, respectively ([§2.3.2.2](#)).

In PowerShell, `bool` maps to `System.Boolean`.

4.2.2 Character

A character value has type `char`, which is capable of storing any UTF-16-encoded 16-bit Unicode code point.

The type `char` has the following accessible members:

[+] [Expand table](#)

Member	Member Kind	Type	Purpose
MaxValue	Static property (read-only)	char	The largest possible value of type <code>char</code>
MinValue	Static property (read-only)	char	The smallest possible value of type <code>char</code>
IsControl	Static method	bool/char	Tests if the character is a control character
IsDigit	Static method	bool/char	Tests if the character is a decimal digit
IsLetter	Static method	bool/char	Tests if the character is an alphabetic letter

Member	Member Kind	Type	Purpose
IsLetterOrDigit	Static method	bool/char	Tests if the character is a decimal digit or alphabetic letter
IsLower	Static method	bool/char	Tests if the character is a lowercase alphabetic letter
IsPunctuation	Static method	bool/char	Tests if the character is a punctuation mark
IsUpper	Static method	bool/char	Tests if the character is an uppercase alphabetic letter
IsWhiteSpace	Static method	bool/char	Tests if the character is a white space character.
ToLower	Static method	char/string	Converts the character to lowercase
ToUpper	Static method	char/string	Converts the character to uppercase

Windows PowerShell: char maps to System.Char.

4.2.3 Integer

There are two signed integer types, both of which use two's-complement representation for negative values:

- Type `int`, which uses 32 bits giving it a range of -2147483648 to +2147483647, inclusive.
- Type `long`, which uses 64 bits giving it a range of -9223372036854775808 to +9223372036854775807, inclusive.

Type `int` has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
.MaxValue	Static property (read-only)	int	The largest possible value of type <code>int</code>
..MinValue	Static property (read-only)	int	The smallest possible value of type <code>int</code>

Type `long` has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
.MaxValue	Static property (read-only)	long	The largest possible value of type long
..MinValue	Static property (read-only)	long	The smallest possible value of type long

There is one unsigned integer type:

- Type `byte`, which uses 8 bits giving it a range of 0 to 255, inclusive.

Type `byte` has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
..MaxValue	Static property (read-only)	byte	The largest possible value of type byte
..MinValue	Static property (read-only)	byte	The smallest possible value of type byte

In PowerShell, `byte`, `int`, and `long` map to `System.Byte`, `System.Int32`, and `System.Int64`, respectively.

4.2.4 Real number

4.2.4.1 float and double

There are two real (or floating-point) types:

- Type `float` uses the 32-bit IEEE single-precision representation.
- Type `double` uses the 64-bit IEEE double-precision representation.

A third type name, `single`, is a synonym for type `float`; `float` is used throughout this specification.

Although the size and representation of the types `float` and `double` are defined by this specification, an implementation may use extended precision for intermediate results.

Type `float` has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
..MaxValue	Static property (read-only)	float	The largest possible value of type float

Member	Member Kind	Type	Purpose
MinValue	Static property (read-only)	float	The smallest possible value of type float
NaN	Static property (read-only)	float	The constant value Not-a-Number
NegativeInfinity	Static property (read-only)	float	The constant value negative infinity
PositiveInfinity	Static property (read-only)	float	The constant value positive infinity

Type double has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
.MaxValue	Static property (read-only)	double	The largest possible value of type double
.MinValue	Static property (read-only)	double	The smallest possible value of type double
.NaN	Static property (read-only)	double	The constant value Not-a-Number
.NegativeInfinity	Static property (read-only)	double	The constant value negative infinity
.PositiveInfinity	Static property (read-only)	double	The constant value positive infinity

In PowerShell, `float` and `double` map to `System.Single` and `System.Double`, respectively.

4.2.4.2 decimal

Type decimal uses a 128-bit representation. At a minimum it must support a scale s such that $0 \leq s \leq$ at least 28, and a value range -79228162514264337593543950335 to 79228162514264337593543950335. The actual representation of decimal is implementation defined.

Type decimal has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
.MaxValue	Static property (read-only)	decimal	The largest possible value of type decimal
..MinValue	Static property (read-only)	decimal	The smallest possible value of type decimal

ⓘ Note

Decimal real numbers have a characteristic called *scale*, which represents the number of digits to the right of the decimal point. For example, the value 2.340 has a scale of 3 where trailing zeros are significant. When two decimal real numbers are added or subtracted, the scale of the result is the larger of the two scales. For example, $1.0 + 2.000$ is 3.000, while $5.0 - 2.00$ is 3.00. When two decimal real numbers are multiplied, the scale of the result is the sum of the two scales. For example, $1.0 * 2.000$ is 2.0000. When two decimal real numbers are divided, the scale of the result is the scale of the first less the scale of the second. For example, $4.00000/2.000$ is 2.00. However, a scale cannot be less than that needed to preserve the correct result. For example, $3.000/2.000$, $3.00/2.000$, $3.0/2.000$, and $3/2$ are all 1.5.

In PowerShell, `decimal` maps to `System.Decimal`. The representation of decimal is as follows:

- When considered as an array of four `int` values it contains the following elements:
 - Index 0 (bits 0-31) contains the low-order 32 bits of the decimal's coefficient.
 - Index 1 (bits 32-63) contains the middle 32 bits of the decimal's coefficient.
 - Index 2 (bits 64-95) contains the high-order 32 bits of the decimal's coefficient.
 - Index 3 (bits 96-127) contains the sign bit and scale, as follows:
 - bits 0--15 are zero
 - bits 16-23 contains the scale as a value 0--28
 - bits 24-30 are zero
 - bit 31 is the sign (0 for positive, 1 for negative)

4.2.5 The switch type

This type is used to constrain the type of a parameter in a command ([§8.10.5](#)). If an argument having the corresponding parameter name is present the parameter tests `$true`; otherwise, it tests `$false`.

In PowerShell, `switch` maps to `System.Management.Automation.SwitchParameter`.

4.2.6 Enumeration types

An enumeration type is one that defines a set of named constants representing all the possible values that can be assigned to an object of that enumeration type. In some cases, the set of values are such that only one value can be represented at a time. In other cases, the set of values are distinct powers of two, and by using the -bor operator ([§7.8.5](#)), multiple values can be encoded in the same object.

The PowerShell environment provides a number of enumeration types, as described in the following sections.

4.2.6.1 Action-Preference type

This implementation-defined type has the following mutually exclusive-valued accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Purpose
Continue	Enumeration constant	The PowerShell runtime will continue processing and notify the user that an action has occurred.
Inquire	Enumeration constant	The PowerShell runtime will stop processing and ask the user how it should proceed.
SilentlyContinue	Enumeration constant	The PowerShell runtime will continue processing without notifying the user that an action has occurred.
Stop	Enumeration constant	The PowerShell runtime will stop processing when an action occurs.

In PowerShell, this type is `System.Management.Automation.ActionPreference`.

4.2.6.2 Confirm-Impact type

This implementation-defined type has the following mutually exclusive-valued accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Purpose
High	Enumeration constant	The action performed has a high risk of losing data, such as reformatting a hard disk.

Member	Member Kind	Purpose
Low	Enumeration constant	The action performed has a low risk of losing data.
Medium	Enumeration constant	The action performed has a medium risk of losing data.
None	Enumeration constant	Do not confirm any actions (suppress all requests for confirmation).

In PowerShell, this type is `System.Management.Automation.ConfirmImpact`.

4.2.6.3 File-Attributes type

This implementation-defined type has the following accessible members, which can be combined:

[\[+\] Expand table](#)

Member	Member Kind	Purpose
Archive	Enumeration constant	The file's archive status. Applications use this attribute to mark files for backup or removal.
Compressed	Enumeration constant	The file is compressed.
Device		Reserved for future use.
Directory	Enumeration constant	The file is a directory.
Encrypted	Enumeration constant	The file or directory is encrypted. For a file, this means that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and directories.
Hidden	Enumeration constant	The file is hidden, and thus is not included in an ordinary directory listing.
Normal	Enumeration constant	The file is normal and has no other attributes set. This attribute is valid only if used alone.
NotContentIndexed	Enumeration constant	The file will not be indexed by the operating system's content indexing service.
Offline	Enumeration constant	The file is offline. The data of the file is not immediately available.

Member	Member Kind	Purpose
ReadOnly	Enumeration constant	The file is read-only.
ReparsePoint	Enumeration constant	The file contains a reparse point, which is a block of user-defined data associated with a file or a directory.
SparseFile	Enumeration constant	The file is a sparse file. Sparse files are typically large files whose data are mostly zeros.
System	Enumeration constant	The file is a system file. The file is part of the operating system or is used exclusively by the operating system.
Temporary	Enumeration constant	The file is temporary. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

In PowerShell, this type is `System.IO.FileAttributes` with attribute `FlagsAttribute`.

4.2.6.4 Regular-Expression-Option type

This implementation-defined type has the following accessible members, which can be combined:

[\[+\] Expand table](#)

Member	Member Kind	Purpose
IgnoreCase	Enumeration constant	Specifies that the matching is case-insensitive.
None	Enumeration constant	Specifies that no options are set.

An implementation may provide other values.

In PowerShell, this type is `System.Text.RegularExpressions.RegexOptions` with attribute `FlagsAttribute`. The following extra values are defined: `Compiled`, `CultureInvariant`, `ECMAScript`, `ExplicitCapture`, `IgnorePatternWhitespace`, `Multiline`, `RightToLeft`, `Singleline`.

4.3 Reference types

4.3.1 Strings

A string value has type `string` and is an immutable sequence of zero or more characters of type `char` each containing a UTF-16-encoded 16-bit Unicode code point.

Type `string` has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
Length	Instance Property	int (read-only)	Gets the number of characters in the string
ToLower	Instance Method	string	Creates a new string that contains the lowercase equivalent
ToUpper	Instance Method	string	Creates a new string that contains the uppercase equivalent

In PowerShell, `string` maps to `System.String`.

4.3.2 Arrays

All array types are derived from the type `array`. This type has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
Length	Instance Property (read-only)	int	Number of elements in the array
Rank	Instance Property (read-only)	int	Number of dimensions in the array
Copy	Static Method	void/see Purpose column	Copies a range of elements from one array to another. There are four versions, where <code>source</code> is the source array, <code>destination</code> is the destination array, <code>count</code> is the number of elements to copy, and <code>sourceIndex</code> and <code>destinationIndex</code> are the starting locations in their respective arrays: <code>Copy(source, destination, int count)</code> <code>Copy(source, destination, long count)</code> <code>Copy(source, sourceIndex, destination, destinationIndex,</code>

Member	Member Kind	Type	Purpose
		int <i>count</i>) Copy(<i>source</i> , <i>sourceIndex</i> , <i>destination</i> , <i>destinationIndex</i> , long <i>count</i>)	
GetLength	Instance Method (read-only)	int/none	Number of elements in a given dimension GetLength(int <i>dimension</i>)

For more details on arrays, see [§9](#).

In PowerShell, `array` maps to `System.Array`.

4.3.3 Hashtables

Type `Hashtable` has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
Count	Instance Property	int	Gets the number of key/value pairs in the Hashtable
Keys	Instance Property	Implementation-defined	Gets a collection of all the keys
Values	Instance Property	Implementation-defined	Gets a collection of all the values
Remove	Instance Method	void/none	Removes the designated key/value

For more details on Hashtables, see [§10](#).

In PowerShell, `Hashtable` maps to `System.Collections.Hashtable`. `Hashtable` elements are stored in an object of type `DictionaryEntry`, and the collections returned by `Keys` and `Values` have type `ICollection`.

4.3.4 The `xml` type

Type `xml` implements the W3C Document Object Model (DOM) Level 1 Core and the Core DOM Level 2. The DOM is an in-memory (cache) tree representation of an XML

document and enables the navigation and editing of this document. This type supports the subscript operator [] ([§7.1.4.4](#)).

In PowerShell, `xml` maps to `System.Xml.XmlDocument`.

4.3.5 The regex type

Type `regex` provides machinery for supporting regular expression processing. It is used to constrain the type of a parameter ([§5.3](#)) whose corresponding argument might contain a regular expression.

In PowerShell, `regex` maps to `System.Text.RegularExpressions.Regex`.

4.3.6 The ref type

Ordinarily, arguments are passed to commands by value. In the case of an argument having some value type a copy of the value is passed. In the case of an argument having some reference type a copy of the reference is passed.

Type `ref` provides machinery to allow arguments to be passed to commands by reference, so the commands can modify the argument's value. Type `ref` has the following accessible members:

[+] [Expand table](#)

Member	Member Kind	Type	Purpose
Value	Instance property (read-write)	The type of the value being referenced.	Gets/sets the value being referenced.

Consider the following function definition and call:

```
PowerShell

function Doubler {
    param ([ref]$x) # parameter received by reference
    $x.Value *= 2.0 # note that 2.0 has type double
}

$number = 8 # designates a value of type int, value 8
Doubler([ref]$number) # argument received by reference
$number # designates a value of type double, value 8.0
```

Consider the case in which `$number` is type-constrained:

PowerShell

```
[int]$number = 8 # designates a value of type int, value 8
Doubler([ref]$number) # argument received by reference
$number # designates a value of type int, value 8
```

As shown, both the argument and its corresponding parameter must be declared `ref`.

In PowerShell, `ref` maps to `System.Management.Automation.PSReference`.

4.3.7 The scriptblock type

Type `scriptblock` represents a precompiled block of script text ([§7.1.8](#)) that can be used as a single unit. It has the following accessible members:

[] [Expand table](#)

Member	Member Kind	Type	Purpose
Attributes	Instance property (read-only)	Collection of attributes	Gets the attributes of the script block.
File	Instance property (read-only)	string	Gets the name of the file in which the script block is defined.
Module	Instance property (read-only)	implementation defined ([§4.5.12] [§4.5.12])	Gets information about the module in which the script block is defined.
GetNewClosure	Instance method	scriptblock /none	Retrieves a script block that is bound to a module. Any local variables that are in the context of the caller will be copied into the module.
Invoke	Instance method	Collection of object/object[]	Invokes the script block with the specified arguments and returns the results.
InvokeReturnAsIs	Instance method	object/object[]	Invokes the script block with the specified arguments and returns any objects generated.
Create	Static method	scriptblock /string	Creates a new scriptblock object that contains the specified script.

In PowerShell, `scriptblock` maps to `System.Management.Automation.ScriptBlock`. `Invoke` returns a collection of `PSObject`.

4.3.8 The math type

Type `math` provides access to some constants and methods useful in mathematical computations. It has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
E	Static property (read-only)	double	Natural logarithmic base
PI	Static property (read-only)	double	Ratio of the circumference of a circle to its diameter
Abs	Static method	numeric/numeric	Absolute value (the return type is the same as the type of the argument passed in)
Acos	Static method	double / double	Angle whose cosine is the specified number
Asin	Static method	double / double	Angle whose sine is the specified number
Atan	Static method	double / double	Angle whose tangent is the specified number
Atan2	Static method	double / double <i>y</i> , double <i>x</i>	Angle whose tangent is the quotient of two specified numbers <i>x</i> and <i>y</i>
Ceiling	Static method	decimal / decimal double / double	smallest integer greater than or equal to the specified number
Cos	Static method	double / double	Cosine of the specified angle
Cosh	Static method	double / double	Hyperbolic cosine of the specified angle
Exp	Static method	double / double	e raised to the specified power
Floor	Static method	decimal / decimal double / double	Largest integer less than or equal to the specified number
Log	Static method	double / double <i>number</i>	Logarithm of number using base e or base <i>base</i>

Member	Member Kind	Type	Purpose
		double / double <i>number</i> , double <i>base</i>	
Log10	Static method	double / double	Base-10 logarithm of a specified number
Max	Static method	numeric/numeric	Larger of two specified numbers (the return type is the same as the type of the arguments passed in)
Min	Static method	numeric/numeric, numeric	Smaller of two specified numbers (the return type is the same as the type of the arguments passed in)
Pow	Static method	double / double <i>x</i> , double <i>y</i>	A specified number <i>x</i> raised to the specified power <i>y</i>
Sin	Static method	double / double	Sine of the specified angle
Sinh	Static method	double / double	Hyperbolic sine of the specified angle
Sqrt	Static method	double / double	Square root of a specified number
Tan	Static method	double / double	Tangent of the specified angle
Tanh	Static method	double / double	Hyperbolic tangent of the specified angle

In PowerShell, `Math` maps to `System.Math`.

4.3.9 The ordered type

Type `ordered` is a pseudo type used only for conversions.

4.3.10 The pscustomobject type

Type `pscustomobject` is a pseudo type used only for conversions.

4.4 Generic types

A number of programming languages and environments provide types that can be *specialized*. Many of these types are referred to as *container types*, as instances of them are able to contain objects of some other type. Consider a type called Stack that can represent a stack of values, which can be pushed on and popped off. Typically, the user of a stack wants to store only one kind of object on that stack. However, if the language or environment does not support type specialization, multiple distinct variants of the

type Stack must be implemented even though they all perform the same task, just with different type elements.

Type specialization allows a *generic type* to be implemented such that it can be constrained to handling some subset of types when it is used. For example,

- A generic stack type that is specialized to hold strings might be written as `Stack[string]`.
- A generic dictionary type that is specialized to hold int keys with associated string values might be written as `Dictionary[int,string]`.
- A stack of stack of strings might be written as `Stack[Stack[string]]`.

Although PowerShell does not define any built-in generic types, it can use such types if they are provided by the host environment. See the syntax in [§7.1.10](#).

The complete name for the type `Stack[string]` suggested above is

`System.Collections.Generic.Stack[string]`. The complete name for the type

`Dictionary[int,string]` suggested above is

`System.Collections.Generic.Dictionary[int,string]`.

4.5 Anonymous types

In some circumstances, an implementation of PowerShell creates objects of some type, and those objects have members accessible to script. However, the actual name of those types need not be specified, so long as the accessible members are specified sufficiently for them to be used. That is, scripts can save objects of those types and access their members without actually knowing those types' names. The following subsections specify these types.

4.5.1 Provider description type

This type encapsulates the state of a provider. It has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
Drives	Instance property (read-only)	Implementation defined (§4.5.2)	A collection of drive description objects
Name	Instance property (read-only)	string	The name of the provider

In PowerShell, this type is `System.Management.Automation.ProviderInfo`.

4.5.2 Drive description type

This type encapsulates the state of a drive. It has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
CurrentLocation	Instance property (read-write)	string	The current working location (§3.1.4) of the drive
Description	Instance property (read-write)	string	The description of the drive
Name	Instance property (read-only)	string	The name of the drive
Root	Instance property (read-only)	string	The name of the drive

In PowerShell, this type is `System.Management.Automation.PSDriveInfo`.

4.5.3 Variable description type

This type encapsulates the state of a variable. It has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
Attributes	Instance property (read-only)	Implementation defined	A collection of attributes
Description	Instance property (read-write)	string	The description assigned to the variable via the New-Variable or Set-Variable cmdlets.
Module	Instance property (read-only)	Implementation defined (§4.5.12)	The module from which this variable was exported
ModuleName	Instance property (read-only)	string	The module in which this variable was defined

Member	Member Kind	Type	Purpose
Name	Instance property (read-only)	string	The name assigned to the variable when it was created in the PowerShell language or via the <code>New-Variable</code> and <code>Set-Variable</code> cmdlets.
Options	Instance property (read-write)	string	The options assigned to the variable via the <code>New-Variable</code> and <code>Set-Variable</code> cmdlets.
Value	Instance property (read-write)	object	The value assigned to the variable when it was assigned in the PowerShell language or via the <code>New-Variable</code> and <code>Set-Variable</code> cmdlets.

In PowerShell, this type is `System.Management.Automation.PSVariable`.

Windows PowerShell: The type of the attribute collection is `System.Management.Automation.PSVariableAttributeCollection`.

4.5.4 Alias description type

This type encapsulates the state of an alias. It has the following accessible members:

[Expand table](#)

Member	Member Kind	Type	Purpose
CommandType	Instance property (read-only)	Implementation defined	Should compare equal with "Alias".
Definition	Instance property (read-only)	string	The command or alias to which the alias was assigned via the <code>New-Alias</code> or <code>Set-Alias</code> cmdlets.
Description	Instance property (read-write)	string	The description assigned to the alias via the <code>New-Alias</code> or <code>Set-Alias</code> cmdlets.
Module	Instance property (read-only)	Implementation defined (§4.5.12)	The module from which this alias was exported
ModuleName	Instance property (read-only)	string	The module in which this alias was defined

Member	Member Kind	Type	Purpose
Name	Instance property (read-only)	string	The name assigned to the alias when it was created via the <code>New-Alias</code> or <code>Set-Alias</code> cmdlets.
Options	Instance property (read-write)	string	The options assigned to the alias via the New-Alias <code>New-Alias</code> or <code>Set-Alias</code> cmdlets.
OutputType	Instance property (read-only)	Implementation defined collection	Specifies the types of the values output by the command to which the alias refers.
Parameters	Instance property (read-only)	Implementation defined collection	The parameters of the command.
ParameterSets	Instance property (read-only)	Implementation defined collection	Information about the parameter sets associated with the command.
ReferencedCommand	Instance property (read-only)	Implementation defined	Information about the command that is immediately referenced by this alias.
ResolvedCommand	Instance property (read-only)	Implementation defined	Information about the command to which the alias eventually resolves.

In PowerShell, this type is `System.Management.Automation.AliasInfo`.

4.5.5 Working location description type

This type encapsulates the state of a working location. It has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
Drive	Instance property (read-only)	Implementation defined (§4.5.2)	A drive description object
Path	Instance property (read-only)	string	The working location
Provider	Instance property	Implementation defined	The provider

Member	Member Kind	Type	Purpose
	(read-only)	(§4.5.1)	
ProviderPath	Instance property (read-only)	string	The current path of the provider

A stack of working locations is a collection of working location objects, as described above.

In PowerShell, a current working location is represented by an object of type `System.Management.Automation.PathInfo`. A stack of working locations is represented by an object of type `System.Management.Automation.PathInfoStack`, which is a collection of `PathInfo` objects.

4.5.6 Environment variable description type

This type encapsulates the state of an environment variable. It has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
Name	Instance property (read-write)	string	The name of the environment variable
Value	Instance property (read-write)	string	The value of the environment variable

In PowerShell, this type is `System.Collections.DictionaryEntry`. The name of the variable is the dictionary key. The value of the environment variable is the dictionary value. `Name` is an `AliasProperty` that equates to `Key`.

4.5.7 Application description type

This type encapsulates the state of an application. It has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
CommandType	Instance property (read-only)	Implementation defined	Should compare equal with "Application".

Member	Member Kind	Type	Purpose
Definition	Instance property (read-only)	string	A description of the application.
Extension	Instance property (read-write)	string	The extension of the application file.
Module	Instance property (read-only)	Implementation defined (§4.5.12)	The module that defines this command.
ModuleName	Instance property (read-only)	string	The name of the module that defines the command.
Name	Instance property (read-only)	string	The name of the command.
OutputType	Instance property (read-only)	Implementation defined collection	Specifies the types of the values output by the command.
Parameters	Instance property (read-only)	Implementation defined collection	The parameters of the command.
ParameterSets	Instance property (read-only)	Implementation defined collection	Information about the parameter sets associated with the command.
Path	Instance property (read-only)	string	Gets the path of the application file.

In PowerShell, this type is `System.Management.Automation.ApplicationInfo`.

4.5.8 Cmdlet description type

This type encapsulates the state of a cmdlet. It has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
CommandType	Instance property (read-only)	Implementation defined	Should compare equal with "Cmdlet".
DefaultParameterSet	Instance property (read-only)	Implementation defined	The default parameter set that is used if PowerShell cannot determine which parameter set to use based on the supplied arguments.
Definition	Instance property (read-only)	string	A description of the cmdlet.
HelpFile	Instance property (read-write)	string	The path to the Help file for the cmdlet.
ImplementingType	Instance property (read-write)	Implementation defined	The type that implements the cmdlet.
Module	Instance property (read-only)	Implementation defined (§4.5.12)	The module that defines this cmdlet.
ModuleName	Instance property (read-only)	string	The name of the module that defines the cmdlet.
Name	Instance property (read-only)	string	The name of the cmdlet.
Noun	Instance property (read-only)	string	The noun name of the cmdlet.
OutputType	Instance property (read-only)	Implementation defined collection	Specifies the types of the values output by the cmdlet.
Parameters	Instance property (read-only)	Implementation defined collection	The parameters of the cmdlet.
ParameterSets	Instance property (read-only)	Implementation defined collection	Information about the parameter sets associated with the cmdlet.

Member	Member Kind	Type	Purpose
Verb	Instance property (read-only)	string	The verb name of the cmdlet.
PSSnapIn	Instance property (read-only)	Implementation defined	Windows PowerShell: Information about the Windows PowerShell snap-in that is used to register the cmdlet.

In PowerShell, this type is `System.Management.Automation.CmdletInfo`.

4.5.9 External script description type

This type encapsulates the state of an external script (one that is directly executable by PowerShell, but is not built-in). It has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
CommandType	Instance property (read-only)	Implementation defined	Should compare equal with "ExternalScript".
Definition	Instance property (read-only)	string	A definition of the script.
Module	Instance property (read-only)	Implementation defined (§4.5.12)	The module that defines this script.
ModuleName	Instance property (read-only)	string	The name of the module that defines the script.
Name	Instance property (read-only)	string	The name of the script.
OriginalEncoding	Instance property (read-only)	Implementation defined	The original encoding used to convert the characters of the script to bytes.

Member	Member Kind	Type	Purpose
OutputType	Instance property (read-only)	Implementation defined collection	Specifies the types of the values output by the script.
Parameters	Instance property (read-only)	Implementation defined collection	The parameters of the script.
ParameterSets	Instance property (read-only)	Implementation defined collection	Information about the parameter sets associated with the script.
Path	Instance property (read-only)	string	The path to the script file.
ScriptBlock	Instance property (read-only)	scriptblock	The external script.
ScriptContents	Instance property (read-only)	string	The original contents of the script.

In PowerShell, this type is `System.Management.Automation.ExternalScriptInfo`.

4.5.10 Function description type

This type encapsulates the state of a function. It has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
CmdletBinding	Instance property (read-only)	bool	Indicates whether the function uses the same parameter binding that compiled cmdlets use (see §12.3.5).
CommandType	Instance property (read-only)	Implementation defined	Can be compared for equality with "Function" or "Filter" to see which of those this object represents.
DefaultParameterSet	Instance property	string	Specifies the parameter set to use if that cannot be determined from the

Member	Member Kind	Type	Purpose
	(read-only)		arguments (see §12.3.5).
Definition	Instance property (read-only)	string	A string version of ScriptBlock
Description	Instance property (read-write)	string	The description of the function.
Module	Instance property (read-only)	Implementation defined (§4.5.12)	The module from which this function was exported
ModuleName	Instance property (read-only)	string	The module in which this function was defined
Name	Instance property (read-only)	string	The name of the function
Options	Instance property (read-write)	Implementation defined	The scope options for the function (§3.5.4).
OutputType	Instance property (read-only)	Implementation defined collection	Specifies the types of the values output, in order (see §12.3.6).
Parameters	Instance property (read-only)	Implementation defined collection	Specifies the parameter names, in order. If the function acts like a cmdlet (see CmdletBinding above) the common parameters are included at the end of the collection.
ParameterSets	Instance property (read-only)	Implementation defined collection	Information about the parameter sets associated with the command. For each parameter, the result shows the parameter name and type, and indicates

Member	Member Kind	Type	Purpose
			if the parameter is mandatory, by position or a switch parameter. If the function acts like a cmdlet (see CmdletBinding above) the common parameters are included at the end of the collection.
ScriptBlock	Instance property (read-only)	scriptblock (§4.3.6)	The body of the function

In PowerShell, this type is `System.Management.Automation.FunctionInfo`.

- `CommandType` has type `System.Management.Automation.CommandTypes`.
- `Options` has type `System.Management.Automation.ScopedItemOptions`.
- `OutputType` has type
`System.Collections.ObjectModel.ReadOnlyCollection`1[[System.Management.Automation.PSTypeName, System.Management.Automation]]`.
- `Parameters` has type
`System.Collections.Generic.Dictionary`2[[System.String, mscorelib], [System.Management.Automation.ParameterMetadata, System.Management.Automation]]`
- `System.Collections.ObjectModel.ReadOnlyCollection`1[[System.Management.Automation.CommandParameterSetInfo, System.Management.Automation]]`.
- `Visibility` has type `System.Management.Automation.SessionStateEntryVisibility`.
- PowerShell also has a property called `Visibility`.

4.5.11 Filter description type

This type encapsulates the state of a filter. It has the same set of accessible members as the function description type ([§4.5.10](#)).

In PowerShell, this type is `System.Management.Automation.FilterInfo`. It has the same set of properties as `System.Management.Automation.FunctionInfo` ([§4.5.11](#)).

4.5.12 Module description type

This type encapsulates the state of a module. It has the following accessible members:

[\[\] Expand table](#)

Member	Member Kind	Type	Purpose
Description	Instance property (read-write)	string	The description of the module (set by the manifest)
ModuleType	Instance property (read-only)	Implementation defined	The type of the module (Manifest, Script, or Binary)
Name	Instance property (read-only)	string	The name of the module
Path	Instance property (read-only)	string	The module's path

In PowerShell, this type is `System.Management.Automation.PSModuleInfo`. The type of `ModuleType` is `System.Management.Automation.ModuleType`.

4.5.13 Custom object description type

This type encapsulates the state of a custom object. It has no accessible members.

In PowerShell, this type is `System.Management.Automation.PSCustomObject`. The cmdlets `Import-Module` and `New-Object` can generate an object of this type.

4.5.14 Command description type

The automatic variable `$PSCmdlet` is an object that represents the cmdlet or function being executed. The type of this object is implementation defined; it has the following accessible members:

[\[\] Expand table](#)

Member	Member Kind	Type	Purpose
ParameterSetName	Instance property (read-only)	string	Name of the current parameter set (see <code>ParameterSetName</code>)
ShouldContinue	Instance method	Overloaded /bool	Requests confirmation of an operation from the user.
ShouldProcess	Instance method	Overloaded	Requests confirmation from the user before an operation is performed.

Member	Member Kind	Type	Purpose
		/bool	

In PowerShell, this type is `System.Management.Automation.PSScriptCmdlet`.

4.5.15 Error record description type

The automatic variable `$Error` contains a collection of error records that represent recent errors ([§3.12](#)). Although the type of this collection is unspecified, it does support subscripting to get access to individual error records.

In PowerShell, the collection type is `System.Collections.ArrayList`. The type of an individual error record in the collection is `System.Management.Automation.ErrorRecord`. This type has the following public properties:

- `CategoryInfo` - Gets information about the category of the error.
- `ErrorDetails` - Gets and sets more detailed error information, such as a replacement error message.
- `Exception` - Gets the exception that is associated with this error record.
- `FullyQualifiedErrorId` - Gets the fully qualified error identifier for this error record.
- `InvocationInfo` - Gets information about the command that was invoked when the error occurred.
- `PipelineIterationInfo` - Gets the status of the pipeline when this error record was created
- `TargetObject` - Gets the object that was being processed when the error occurred.

4.5.16 Enumerator description type

A number of variables are enumerators for collections ([§4](#)). The automatic variable `$foreach` is the enumerator created for any `foreach` statement. The automatic variable `$input` is the enumerator for a collection delivered to a function from the pipeline. The automatic variable `$switch` is the enumerator created for any `switch` statement.

The type of an enumerator is implementation defined; it has the following accessible members:

[] [Expand table](#)

Member	Member Kind	Type	Purpose
Current	Instance property (read-only)	object	Gets the current element in the collection. If the enumerator is not currently positioned at an element of the collection, the behavior is implementation defined.
MoveNext	Instance method	None/bool	Advances the enumerator to the next element of the collection. Returns \$true if the enumerator was successfully advanced to the next element; \$false if the enumerator has passed the end of the collection.

In PowerShell, these members are defined in the interface `System.IEnumerator`, which is implemented by the types identified below. If the enumerator is not currently positioned at an element of the collection, an exception of type `InvalidOperationException` is raised. For `$foreach`, this type is `System.Array+SZArrayEnumerator`. For `$input`, this type is `System.Collections.ArrayList+ArrayListEnumeratorSimple`. For `$switch`, this type is `System.Array+SZArrayEnumerator`.

4.5.17 Directory description type

The cmdlet [New-Item](#) can create items of various kinds including FileSystem directories. The type of a directory description object is implementation defined; it has the following accessible members:

[+] [Expand table](#)

Member	Member Kind	Type	Purpose
Attributes	Instance property (read-write)	Implementation defined (§4.2.6.3)	Gets or sets one or more of the attributes of the directory object.
CreationTime	Instance property (read-write)	Implementation defined (§4.5.19)	Gets and sets the creation time of the directory object.
Extension	Instance property (read- only)	string	Gets the extension part of the directory name.
FullName	Instance property (read-only)	string	Gets the full path of the directory.
LastWriteTime	Instance property (read-write)	Implementation defined (§4.5.19)	Gets and sets the time when the directory was last written to.
Name	Instance property (read- only)	string	Gets the name of the directory.

In PowerShell, this type is `System.IO.DirectoryInfo`. The type of the **Attributes** property is `System.IO.FileAttributes`.

4.5.18 File description type

The cmdlet `New-Item` can create items of various kinds including FileSystem files. The type of a file description object is implementation defined; it has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
Attributes	Instance property (read-write)	Implementation defined (§4.2.6.3)	Gets or sets one or more of the attributes of the file object.
BaseName	Instance property (read-only)	string	Gets the name of the file excluding the extension.
CreationTime	Instance property (read-write)	Implementation defined (§4.5.19)	Gets and sets the creation time of the file object.
Extension	Instance property (read-only)	string	Gets the extension part of the file name.
FullName	Instance property (read-only)	string	Gets the full path of the file.
LastWriteTime	Instance property (read-write)	Implementation defined (§4.5.19)	Gets and sets the time when the file was last written to.
Length	Instance property (read-only)	long	Gets the size of the file, in bytes.
Name	Instance property (read-only)	string	Gets the name of the file.
VersionInfo	Instance property	Implementation defined	Windows PowerShell: This ScriptProperty returns a <code>System.DiagnosticsFileVersionInfo</code>

Member	Member Kind	Type	Purpose
	(read-only)		for the file.

In PowerShell, this type is `System.IO.FileInfo`.

4.5.19 Date-Time description type

The type of a date-time description object is implementation defined; it has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
Day	Instance property (read-only)	int	Gets the day component of the month represented by this instance.
Hour	Instance property (read-only)	int	Gets the hour component of the date represented by this instance.
Minute	Instance property (read-only)	int	Gets the minute component of the date represented by this instance.
Month	Instance property (read-only)	int	Gets the month component of the date represented by this instance.
Second	Instance property (read-only)	int	Gets the seconds component of the date represented by this instance.
Year	Instance property (read-only)	int	Gets the year component of the date represented by this instance.

An object of this type can be created by cmdlet [Get-Date](#).

In PowerShell, this type is `System.DateTime`.

4.5.20 Group-Info description type

The type of a **group-info** description object is implementation defined; it has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
Count	Instance property (read-only)	int	Gets the number of elements in the group.
Group	Instance property (read-only)	Implementation-defined collection	Gets the elements of the group.
Name	Instance property (read-only)	string	Gets the name of the group.
Values	Instance property (read-only)	Implementation-defined collection	Gets the values of the elements of the group.

An object of this type can be created by cmdlet [Group-Object](#).

In PowerShell, this type is `Microsoft.PowerShell.Commands.GroupInfo`.

4.5.21 Generic-Measure-Info description type

The type of a **generic-measure-info** description object is implementation defined; it has the following accessible members:

[\[+\] Expand table](#)

Member	Member Kind	Type	Purpose
Average	Instance property (read-only)	double	Gets the average of the values of the properties that are measured.
Count	Instance property (read-only)	int	Gets the number of objects with the specified properties.
Maximum	Instance property (read-only)	double	Gets the maximum value of the specified properties.
Minimum	Instance property (read-only)	double	Gets the minimum value of the specified properties.
Property	Instance property (read-only)	string	Gets the property to be measured.
Sum	Instance property (read-only)	double	Gets the sum of the values of the specified properties.

An object of this type can be created by cmdlet [Measure-Object](#).

In PowerShell, this type is `Microsoft.PowerShell.Commands.GenericMeasureInfo`.

4.5.22 Text-Measure-Info description type

The type of a **text-info** description object is implementation defined; it has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
Characters	Instance property (read-only)	int	Gets the number of characters in the target object.
Lines	Instance property (read-only)	int	Gets the number of lines in the target object.
Property	Instance property (read-only)	string	Gets the property to be measured.
Words	Instance property (read-only)	int	Gets the number of words in the target object.

An object of this type can be created by cmdlet `Measure-Object`.

In PowerShell, this type is `Microsoft.PowerShell.Commands.TextMeasureInfo`.

4.5.23 Credential type

A credential object can then be used in various security operations. The type of a credential object is implementation defined; it has the following accessible members:

[+] Expand table

Member	Member Kind	Type	Purpose
Password	Instance property (read-only)	Implementation defined	Gets the password.
UserName	Instance property (read-only)	string	Gets the username.

An object of this type can be created by cmdlet `Get-Credential`.

In PowerShell, this type is `System.Management.Automation.PSCredential`.

4.5.24 Method designator type

The type of a method designator is implementation defined; it has the following accessible members:

[\[\] Expand table](#)

Member	Member Kind	Type	Purpose
Invoke	Instance method	object/variable number and type	Takes a variable number of arguments, and indirectly calls the method referred to by the parent method designator, passing in the arguments.

An object of this type can be created by an *invocation-expression* ([§7.1.3](#)).

In PowerShell, this type is `System.Management.Automation.PSMethod`.

4.5.25 Member definition type

This type encapsulates the definition of a member. It has the following accessible members:

[\[\] Expand table](#)

Member	Member Kind	Type	Purpose
Definition	Instance property (read-only)	string	Gets the definition of the member.
MemberType	Instance property (read-only)	Implementation defined	Gets the PowerShell type of the member.
Name	Instance property (read-only)	string	Gets the name of the member.
TypeName	Instance property (read-only)	string	Gets the type name of the member.

In PowerShell, this type is `Microsoft.PowerShell.Commands.MemberDefinition`.

4.6 Type extension and adaptation

A PowerShell implementation includes a family of core types (which are documented in this chapter) that each contain their own set of *base members*. Those members can be methods or properties, and they can be instance or static members. For example, the base members of the type `string` ([§4.3.1](#)) are the instance property `Length` and the instance methods `ToLower` and `ToUpper`.

When an object is created, it contains all the instance properties of that object's type, and the instance methods of that type can be called on that object. An object may be customized via the addition of instance members at runtime. The result is called a *custom object*. Any members added to an instance exist only for the life of that instance; other instances of the same core type are unaffected.

The base member set of a type can be augmented by the addition of the following kinds of members:

- *adapted members*, via the *Extended Type System* (ETS), most details of which are unspecified.
- *extended members*, via the cmdlet [Add-Member](#).

In PowerShell, extended members can also be added via `types.ps1xml` files. Adapted and extended members are collectively called **synthetic members**.

The ETS adds the following members to all PowerShell objects: **psbase**, **psadapted**, **psextended**, and **pstypenames**. See the **Force** and **View** parameters in the cmdlet [Get-Member](#) for more information on these members.

An instance member may hide an extended and/or adapted member of the same name, and an extended member may hide an adapted member. In such cases, the member sets **psadapted** and **psextended** can be used to access those hidden members.

If a `types.ps1xml` specifies a member called **Supports**, `obj.psextended` provides access to just that member and not to a member added via [Add-Member](#).

There are three ways create a custom object having a new member M:

1. This approach can be used to add one or more NoteProperty members.

```
PowerShell  
  
$x = New-Object PSObject -Property @{M = 123}
```

2. This approach can be used to add NoteProperty or ScriptMethod members.

```
PowerShell  
  
$x = New-Module -AsCustomObject {$M = 123 ; Export-ModuleMember --Variable M}
```

3. This approach can be used to add any kind of member.

```
PowerShell
```

```
$x = New-Object PSObject  
Add-Member -InputObject $x -Name M -MemberType NoteProperty -Value 123
```

`PSObject` is the base type of all PowerShell types.

5. Variables

Article • 01/08/2025

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

A variable represents a storage location for a value, and that value has a type. Traditional procedural programming languages are statically typed; that is, the runtime type of a variable is that with which it was declared at compile time. Object-oriented languages add the idea of inheritance, which allows the runtime type of a variable to be that with which it was declared at compile time or some type derived from that type. Being a dynamically typed language, PowerShell's variables do not have types, per se. In fact, variables are not defined; they simply come into being when they are first assigned a value. And while a variable may be constrained ([§5.3](#)) to holding a value of a given type, type information in an assignment cannot always be verified statically.

At different times, a variable may be associated with values of different types either through assignment ([§7.11](#)) or the use of the `++` and `--` operators ([§7.1.5](#), [§7.2.6](#)). When the value associated with a variable is changed, that value's type may change. For example,

PowerShell

```
$i = "abc"      # $i holds a value of type string
$i = 2147483647 # $i holds a value of type int
```

```
++$i          # $i now holds a value of type double because  
# 2147483648 is too big to fit in type int
```

Any use of a variable that has not been created results in the value \$null. To see if a variable has been defined, use the [Test-Path](#) cmdlet.

5.1 Writable location

A *writable location* is an expression that designates a resource to which a command has both read and write access. A writable location may be a variable ([§5](#)), an array element ([§9](#)), an associated value in a Hashtable accessed via a subscript ([§10](#)), a property ([§7.1.2](#)), or storage managed by a provider ([§3.1](#)).

5.2 Variable categories

PowerShell defines the following categories of variables: static variables, instance variables, array elements, Hashtable key/value pairs, parameters, ordinary variables, and variables on provider drives. The subsections that follow describe each of these categories.

In the following example

```
PowerShell

function F ($p1, $p2) {
    $radius = 2.45
    $circumference = 2 * ([Math]::PI) * $radius

    $date = Get-Date -Date "2010-2-1 10:12:14 pm"
    $month = $date.Month

    $values = 10, 55, 93, 102
    $value = $values[2]

    $h1 = @{ FirstName = "James"; LastName = "Anderson" }
    $h1.FirstName = "Smith"

    $Alias:A = "Help"
    $Env:MyPath = "E:\Temp"
    ${E:output.txt} = 123
    $Function:F = { "Hello there" }
    $Variable:v = 10
}
```

- `[Math::PI]` is a static variable

- `$date.Month` is an instance variable
- `$values[2]` is an array element
- `$h1.FirstName` is a `Hashtable` key whose corresponding value is `$h1['FirstName']`
- `$p1` and `$p2` are parameters
- `$radius`, `$circumference`, `$date`, `$month`, `$values`, `$value`, and `$h1` are ordinary variables
- `$Alias:A`, `$Env:MyPath`, `${E:output.txt}`, and `$Function:F` are variables on the corresponding provider drives.
- `$Variable:v` is actually an ordinary variable written with its fully qualified provider drive.

5.2.1 Static variables

A data member of an object that belongs to the object's type rather than to that particular instance of the type is called a *static variable*. See [§4.2.3](#), [§4.2.4.1](#), and [§4.3.8](#) for some examples.

PowerShell provides no way to create new types that contain static variables; however, objects of such types may be provided by the host environment.

Memory for creating and deleting objects containing static variables is managed by the host environment and the garbage collection system.

See [§7.1.2](#) for information about accessing a static variable.

A static data member can be a field or a property.

5.2.2 Instance variables

A data member of an object that belongs to a particular instance of the object's type rather than to the type itself is called an *instance variable*. See [§4.3.1](#), [§4.3.2](#), and [§4.3.3](#) for some examples.

A PowerShell host environment might provide a way to create new types that contain instance variables or to add new instance variables to existing types.

Memory for creating and deleting objects containing static variables is managed by the host environment and the garbage collection system.

See [§7.1.2](#) for information about accessing an instance variable.

An instance data member can be a field or a property.

5.2.3 Array elements

An array can be created via a unary comma operator ([§7.2.1](#)), *sub-expression* ([§7.1.6](#)), *array-expression* ([§7.1.7](#)), binary comma operator ([§7.3](#)), range operator ([§7.4](#)), or [New-Object](#) cmdlet.

Memory for creating and deleting arrays is managed by the host environment and the garbage collection system.

Arrays and array elements are discussed in [§9](#).

5.2.4 Hashtable key/value pairs

A Hashtable is created via a hash literal ([§2.3.5.6](#)) or the [New-Object](#) cmdlet. A new key/value pair can be added via the `[]` operator ([§7.1.4.3](#)).

Memory for creating and deleting Hashtables is managed by the host environment and the garbage collection system.

Hashtables are discussed in [§10](#).

5.2.5 Parameters

A parameter is created when its parent command is invoked, and it is initialized with the value of the argument provided in the invocation or by the host environment. A parameter ceases to exist when its parent command terminates.

Parameters are discussed in [§8.10](#).

5.2.6 Ordinary variables

An *ordinary variable* is defined by an *assignment-expression* ([§7.11](#)) or a *foreach-statement* ([§8.4.4](#)). Some ordinary variables are predefined by the host environment while others are transient, coming and going as needed at runtime.

The lifetime of an ordinary variable is that part of program execution during which storage is guaranteed to be reserved for it. This lifetime begins at entry into the scope with which it is associated, and ends no sooner than the end of the execution of that scope. If the parent scope is entered recursively or iteratively, a new instance of the local variable is created each time.

The storage referred to by an ordinary variable is reclaimed independently of the lifetime of that variable.

An ordinary variable can be named explicitly with a **Variable**: namespace prefix ([§5.2.7](#)).

5.2.7 Variables on provider drives

The concept of providers and drives is introduced in [§3.1](#), with each provider being able to provide its own namespace drive(s). This allows resources on those drives to be accessed as though they were ordinary variables ([§5.2.6](#)). In fact, an ordinary variable is stored on the file system provider drive **Variable**: ([§3.1.5](#)) and can be accessed by its ordinary name or its fully qualified namespace name.

Some namespace variable types are constrained implicitly ([§5.3](#)).

5.3 Constrained variables

By default, a variable may designate a value of any type. However, a variable may be *constrained* to designating values of a given type by specifying that type as a type literal before its name in an assignment or a parameter. For example,

PowerShell

```
[int]$i = 10    # constrains $i to designating ints only
$i = "Hello"  # error, no conversion to int
$i = "0x10"   # ok, conversion to int
$i = $true     # ok, conversion to int

function F ([int]$p1, [switch]$p2, [regex]$p3) { ... }
```

Any variable belonging to the namespace **Env**: **Alias**: or to the file system namespace ([§2.3.2](#), [§3.1](#)) is constrained implicitly to the type `string`. Any variable belonging to the namespace **Function**: ([§2.3.2](#), [§3.1](#)) is constrained implicitly to the type `scriptblock`.

6. Conversions

Article • 03/24/2025

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

A *type conversion* is performed when a value of one type is used in a context that requires a different type. If such a conversion happens automatically it is known as *implicit conversion*. (A common example of this is with some operators that need to convert one or more of the values designated by their operands.) Implicit conversion is permitted provided the sense of the source value is preserved, such as no loss of precision of a number when it is converted.

The cast operator ([§7.2.9](#)) allows for *explicit conversion*.

Conversions are discussed below, with supplementary information being provided as necessary in the description of each operator in [§6.19](#).

Explicit conversion of a value to the type it already has causes no change to that value or its representation.

The rules for handing conversion when the value of an expression is being bound to a parameter are covered in [§6.17](#).

6.1 Conversion to void

A value of any type can be discarded explicitly by casting it to type void. There is no result.

6.2 Conversion to bool

The rules for converting any value to type bool are as follows:

- A numeric or char value of zero is converted to False; a numeric or char value of non-zero is converted to True.
- A value of null type is converted to False.
- A string of length 0 is converted to False; a string of length > 0 is converted to True.
- A switch parameter with value `$true` is converted to True, and one with value `$false` is converted to False.
- All other non-null reference type values are converted to True.

If the type implements IList:

- If the object's Length > 2, the value is converted to True.
- If the object's Length is 1 and that first element is not itself an IList, then if that element's value is true, the value is converted to True.
- Otherwise, if the first element's Count ≥ 1 , the value is converted to True.
- Otherwise, the value is converted to False.

6.3 Conversion to char

The rules for converting any value to type char are as follows:

- The conversion of a value of type bool, decimal, float, or double is in error.
- A value of null type is converted to the null (U+0000) character.
- An integer type value whose value can be represented in type char has that value; otherwise, the conversion is in error.
- The conversion of a string value having a length other than 1 is in error.
- A string value having a length 1 is converted to a char having that one character's value.
- A numeric type value whose value after rounding of any fractional part can be represented in the destination type has that rounded value; otherwise, the conversion is in error.
- For other reference type values, if the reference type supports such a conversion, that conversion is used; otherwise, the conversion is in error.

6.4 Conversion to integer

The rules for converting any value to type byte, int, or long are as follows:

- The bool value False is converted to zero; the bool value True is converted to 1.
- A char type value whose value can be represented in the destination type has that value; otherwise, the conversion is in error.
- A numeric type value whose value after rounding of any fractional part can be represented in the destination type has that rounded value; otherwise, the conversion is in error.
- A value of null type is converted to zero.
- A string that represents a number is converted as described in [§6.16](#). If after truncation of the fractional part the result can be represented in the destination type the string is well formed and it has the destination type; otherwise, the conversion is in error. If the string does not represent a number, the conversion is in error.
- For other reference type values, if the reference type supports such a conversion, that conversion is used; otherwise, the conversion is in error.

6.5 Conversion to float and double

The rules for converting any value to type float or double are as follows:

- The bool value False is converted to zero; the bool value True is converted to 1.
- A char value is represented exactly.
- A numeric type value is represented exactly, if possible; however, for int, long, and decimal conversions to float, and for long and decimal conversions to double, some of the least significant bits of the integer value may be lost.
- A value of null type is converted to zero.
- A string that represents a number is converted as described in [§6.16](#); otherwise, the conversion is in error.
- For other reference type values, if the reference type supports such a conversion, that conversion is used; otherwise, the conversion is in error.

6.6 Conversion to decimal

The rules for converting any value to type decimal are as follows:

- The bool value False is converted to zero; the bool value True is converted to 1.
- A char type value is represented exactly.

- A numeric type value is represented exactly; however, if that value is too large or too small to fit in the destination type, the conversion is in error.
- A value of null type is converted to zero.
- A string that represents a number is converted as described in §6.16; otherwise, the conversion is in error.
- For other reference type values, if the reference type supports such a conversion, that conversion is used; otherwise, the conversion is in error.
- The scale of the result of a successful conversion is such that the fractional part has no trailing zeros.

6.7 Conversion to object

The value of any type except the null type (4.1.2) can be converted to type object. The value retains its type and representation.

6.8 Conversion to string

The rules for converting any value to type string are as follows:

- The bool value `$false` is converted to "False"; the bool value `$true` is converted to "True".
- A char type value is converted to a 1-character string containing that char.
- A numeric type value is converted to a string having the form of a corresponding numeric literal. However, the result has no leading or trailing spaces, no leading plus sign, integers have base 10, and there is no type suffix. For a decimal conversion, the scale is preserved. For values of $-\infty$, $+\infty$, and NaN, the resulting strings are "-Infinity", "Infinity", and "NaN", respectively.
- A value of null type is converted to the empty string.
- For a 1-dimensional array, the result is a string containing the value of each element in that array, from start to end, converted to string, with elements being separated by the current Output Field Separator (§2.3.2.2). For an array having elements that are themselves arrays, only the top-level elements are converted. The string used to represent the value of an element that is an array, is implementation defined. For a multi-dimensional array, it is flattened (§9.12) and then treated as a 1-dimensional array.
- A value of null type is converted to the empty string.
- A scriptblock type value is converted to a string containing the text of that block without the delimiting { and } characters.
- For an enumeration type value, the result is a string containing the name of each enumeration constant encoded in that value, separated by commas.

- For other reference type values, if the reference type supports such a conversion, that conversion is used; otherwise, the conversion is in error.

The string used to represent the value of an element that is an array has the form `System.Type[]`, `System.Type[,]`, and so on. For other reference types, the method `ToString` is called. For other enumerable types, the source value is treated like a 1-dimensional array.

6.9 Conversion to array

The rules for converting any value to an array type are as follows:

- The target type may not be a multidimensional array.
- A value of null type is retained as is.
- For a scalar value other than `$null` or a value of type hashtable, a new 1-element array is created whose value is the scalar after conversion to the target element type.
- For a 1-dimensional array value, a new array of the target type is created, and each element is copied with conversion from the source array to the corresponding element in the target array.
- For a multi-dimensional array value, that array is first flattened ([§9.12](#)), and then treated as a 1-dimensional array value.
- A string value is converted to an array of char having the same length with successive characters from the string occupying corresponding positions in the array.

For other enumerable types, a new 1-element array is created whose value is the corresponding element after conversion to the target element type, if such a conversion exists. Otherwise, the conversion is in error.

6.10 Conversion to xml

The object is converted to type string and then into an XML Document object of type `xml`.

6.11 Conversion to regex

An expression that designates a value of type string may be converted to type `regex`.

6.12 Conversion to scriptblock

The rules for converting any value to type `scriptblock` are as follows:

- A string value is treated as the name of a command optionally following by arguments to a call to that command.

6.13 Conversion to enumeration types

The rules for converting any value to an enumeration type are as follows:

- A value of type string that contains one of the named values (with regard for case) for an enumeration type is converted to that named value.
- A value of type string that contains a comma-separated list of named values (with regard for case) for an enumeration type is converted to the bitwise-OR of all those named values.

6.14 Conversion to other reference types

The rules for converting any value to a reference type other than an array type or string are as follows:

- A value of null type is retained as is.
- Otherwise, the behavior is implementation defined.

A number of pieces of machinery come in to play here; these include the possible use of single argument constructors or default constructors if the value is a hashtable, implicit and explicit conversion operators, and Parse methods for the target type; the use of Convert.ConvertTo; and the ETS conversion mechanism.

6.15 Usual arithmetic conversions

If neither operand designates a value having numeric type, then

- If the left operand designates a value of type bool, the conversion is in error.
- Otherwise, all operands designating the value `$null` are converted to zero of type int and the process continues with the numeric conversions listed below.
- Otherwise, if the left operand designates a value of type char and the right operand designates a value of type bool, the conversion is in error.
- Otherwise, if the left operand designates a value of type string but does not represent a number ([§6.16](#)), the conversion is in error.
- Otherwise, if the right operand designates a value of type string but does not represent a number ([§6.16](#)), the conversion is in error.

- Otherwise, all operands designating values of type string are converted to numbers ([§6.16](#)), and the process continues with the numeric conversions listed below.
- Otherwise, the conversion is in error.

Numeric conversions:

- If one operand designates a value of type decimal, the value designated by the other operand is converted to that type, if necessary. The result has type decimal.
- Otherwise, if one operand designates a value of type double, the value designated by the other operand is converted to that type, if necessary. The result has type double.
- Otherwise, if one operand designates a value of type float, the values designated by both operands are converted to type double, if necessary. The result has type double.
- Otherwise, if one operand designates a value of type long, the value designated by the other operand value is converted to that type, if necessary. The result has the type first in the sequence long and double that can represent its value.
- Otherwise, the values designated by both operands are converted to type int, if necessary. The result has the first in the sequence int, long, double that can represent its value without truncation.

6.16 Conversion from string to numeric type

Depending on its contents, a string can be converted explicitly or implicitly to a numeric value. Specifically,

- An empty string is converted to the value zero.
- Leading and trailing spaces are ignored; however, a string may not consist of spaces only.
- A string containing only white space and/or line terminators is converted to the value zero.
- One leading + or - sign is permitted.
- An integer number may have a hexadecimal prefix (0x or 0X).
- An optionally signed exponent is permitted.
- Type suffixes and multipliers are not permitted.
- The case-distinct strings "-Infinity", "Infinity", and "NaN" are recognized as the values $-\infty$, $+\infty$, and NaN, respectively.

6.17 Conversion during parameter binding

For information about parameter binding see [§8.14](#).

When the value of an expression is being bound to a parameter, there are extra conversion considerations, as described below:

- If the parameter type is switch ([§4.2.5](#), [§8.10.5](#)) and the parameter has no argument, the value of the parameter in the called command is set to `$true`. If the parameter type is other than switch, a parameter having no argument is in error.
- If the parameter type is switch and the argument value is `$null`, the parameter value is set to `$false`.
- If the parameter type is object or is the same as the type of the argument, the argument's value is passed without conversion.
- If the parameter type is not object or scriptblock, an argument having type scriptblock is evaluated and its result is passed as the argument's value. (This is known as *delayed script block binding*.) If the parameter type is object or scriptblock, an argument having type scriptblock is passed as is.
- If the parameter type is a collection of type T2, and the argument is a scalar of type T1, that scalar is converted to a collection of type T2 containing one element. If necessary, the scalar value is converted to type T2 using the conversion rules of this section.
- If the parameter type is a scalar type other than object and the argument is a collection, the argument is in error.
- If the expected parameter type is a collection of type T2, and the argument is a collection of type T1, the argument is converted to a collection of type T2 having the same length as the argument collection. If necessary, the argument collection element values are converted to type T2 using the conversion rules of this section.
- If the steps above and the conversions specified earlier in this chapter do not suffice, the rules in [§6.18](#) are applied. If those fail, the parameter binding fails.

6.18 .NET Conversion

For an implicit conversion, PowerShell's built-in conversions are tried first. If they cannot resolve the conversion, the .NET custom converters below are tried, in order, from top to bottom. If a conversion is found, but it throws an exception, the conversion has failed.

- **PSTypeConverter**: There are two ways of associating the implementation of the **PSTypeConverter** class with its target class: through the type configuration file (`types.ps1xml`) or by applying the `System.ComponentModel.TypeConverterAttribute` attribute to the target class. Refer to the PowerShell SDK documentation for more information.

- **TypeConverter**: This CLR type provides a unified way of converting types of values to other types, as well as for accessing standard values and sub-properties. The most common type of converter is one that converts to and from a text representation. The type converter for a class is bound to the class with a `System.ComponentModel.TypeConverterAttribute`. Unless this attribute is overridden, all classes that inherit from this class use the same type converter as the base class. Refer to the PowerShell SDK and the Microsoft .NET framework documentation for more information.
- **Parse Method**: If the source type is string and the destination type has a method called `Parse`, that method is called to perform the conversion.
- **Constructors**: If the destination type has a constructor taking a single argument whose type is that of the source type, that constructor is called to perform the conversion.
- **Implicit Cast Operator**: If the source type has an implicit cast operator that converts to the destination type, that operator is called to perform the conversion.
- **Explicit Cast Operator**: If the source type has an explicit cast operator that converts to the destination type, that operator is called to perform the conversion. If the destination type has an explicit cast operator that converts from the source type, that operator is called to perform the conversion.
- **IConvertible**: `System.Convert.ChangeType` is called to perform the conversion.

6.19 Conversion to ordered

The rules for converting any value to the pseudo-type ordered are as follows:

- If the value is a hash literal ([§2.3.5.6](#)), the result is an object with an implementation defined type that behaves like a hashtable and the order of the keys matches the order specified in the hash literal.
- Otherwise, the behavior is implementation defined.

Only hash literals ([§2.3.5.6](#)) can be converted to ordered. The result is an instance of `System.Collections.Specialized.OrderedDictionary`.

6.20 Conversion to pscustomobject

The rules for converting any value to the pseudo-type pscustomobject are as follows:

- A value of type hashtable is converted to a PowerShell object. Each key in the hashtable becomes a NoteProperty with the corresponding value.
- Otherwise, the behavior is implementation defined.

The conversion is always allowed but does not change the type of the value.

7. Expressions

Article • 04/25/2024

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

Syntax:

```
Syntax

expression:
    primary-expression
    bitwise-expression
    logical-expression
    comparison-expression
    additive-expression
    multiplicative-expression

dash: one of
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)

dashdash:
    dash dash
```

Description:

An *expression* is a sequence of operators and operands that designates a method, a function, a writable location, or a value; specifies the computation of a value; produces one or more side effects; or performs some combination thereof. For example,

- The literal 123 is an expression that designates the int value 123.
- The expression `1,2,3,4` designates the 4-element array object having the values shown.
- The expression `10.4 * $a` specifies a computation.
- The expression `$a++` produces a side effect.
- The expression `$a[$i--] = $b[++$j]` performs a combination of these things.

Except as specified for some operators, the order of evaluation of terms in an expression and the order in which side effects take place are both unspecified. Examples of unspecified behavior include the following: `$i++ + $i`, `$i + --$i`, and `$w[$j++] = $v[$j]`.

An implementation of PowerShell may provide support for user-defined types, and those types may have operations defined on them. All details of such types and operations are implementation defined.

A *top-level expression* is one that is not part of some larger expression. If a top-level expression contains a side-effect operator the value of that expression is not written to the pipeline; otherwise, it is. See [§7.1.1](#) for a detailed discussion of this.

Ordinarily, an expression that designates a collection ([§4§4]) is enumerated into its constituent elements when the value of that expression is used. However, this is not the case when the expression is a cmdlet invocation. For example,

```
PowerShell

$x = 10,20,30
$a = $($x; 99)          # $a.Length is 4

$x = New-Object 'int[]' 3
$a = $($x; 99)          # equivalent, $a.Length is 4

$a = $(New-Object 'int[]' 3; 99)  # $a.Length is 2
```

In the first two uses of the `$(...)` operator, the expression designating the collection is the variable `$x`, which is enumerated resulting in three `int` values, plus the `int` 99. However, in the third case, the expression is a direct call to a cmdlet, so the result is not enumerated, and `$a` is an array of two elements, `int[3]` and `int`.

If an operation is not defined by PowerShell, the type of the value designated by the left operand is inspected to see if it has a corresponding `op_<operation>` method.

7.1 Primary expressions

Syntax:

```
Syntax

primary-expression:
    value
    member-access
    element-access
    invocation-expression
    post-increment-expression
    post-decrement-expression

value:
    parenthesized-expression
    sub-expression
    array-expression
    script-block-expression
    hash-literal-expression
    literal
    type-literal
    variable
```

7.1.1 Grouping parentheses

Syntax:

Tip

The `~opt~` notation in the syntax definitions indicates that the lexical entity is optional in the syntax.

```
Syntax
```

```
parenthesized-expression:
    ( new-lines~opt~ pipeline new-lines~opt~ )
```

Description:

A parenthesized expression is a *primary-expression* whose type and value are the same as those of the expression without the parentheses. If the expression designates a

variable then the parenthesized expression designates that same variable. For example, `$x.m` and `(x).m` are equivalent.

Grouping parentheses may be used in an expression to document the default precedence and associativity within that expression. They can also be used to override that default precedence and associativity. For example,

PowerShell

```
4 + 6 * 2    # 16
4 + (6 * 2)  # 16 document default precedence
(4 + 6) * 2  # 20 override default precedence
```

Ordinarily, grouping parentheses at the top-most level are redundant. However, that is not always the case. Consider the following example:

PowerShell

```
2,4,6      # Length 3; values 2,4,6
(2,4),6    # Length 2; values [Object[],int]
```

In the second case, the parentheses change the semantics, resulting in an array whose two elements are an array of 2 ints and the scalar int 6.

Here's another exception:

PowerShell

```
23.5/2.4      # pipeline gets 9.79166666666667
$a = 1234 * 3.5 # value not written to pipeline
$a            # pipeline gets 4319
```

In the first and third cases, the value of the result is written to the pipeline. However, although the expression in the second case is evaluated, the result is not written to the pipeline due to the presence of the side-effect operator `=` at the top level. (Removal of the `$a =` part allows the value to be written, as `*` is not a side-effect operator.)

To stop a value of any expression not containing top-level side effects from being written to the pipeline, discard it explicitly, as follows:

PowerShell

```
# None of these value are written to pipeline
[void](23.5/2.4)
[void]$a
```

```
$null = $a  
$a > $null
```

To write to the pipeline the value of any expression containing top-level side effects, enclose that expression in parentheses, as follows:

PowerShell

```
($a = 1234 * 3.5) # pipeline gets 4319
```

As such, the grouping parentheses in this case are not redundant.

In the following example, we have variable substitution ([§2.3.5.2](#)) taking place in a string literal:

PowerShell

```
">$($a = -23)<" # value not written to pipeline, get >  
">$(($a = -23))<" # pipeline gets >-23<
```

In the first case, the parentheses represent a *sub-expression's* delimiters *not* grouping parentheses, and as the top-level expression contains a side-effect operator, the expression's value is not written to the pipeline. Of course, the `>` and `<` characters are still written.) If grouping parenthesis are added -- as shown in the second case -- writing is enabled.

The following examples each contain top-level side-effect operators:

PowerShell

```
$a = $b = 0      # value not written to pipeline  
$a = ($b = 0)    # value not written to pipeline  
($a = ($b = 0)) # pipeline gets 0  
  
++$a            # value not written to pipeline  
(++$b)          # pipeline gets 1  
  
$a--            # value not written to pipeline  
($b--)          # pipeline gets 1
```

The use of grouping parentheses around an expression containing no top-level side effects makes those parentheses redundant. For example;

PowerShell

```
$a      # pipeline gets 0  
($a)    # no side effect, so () redundant
```

Consider the following example that has two side effects, neither of which is at the top level:

PowerShell

```
12.6 + ($a = 10 - ++$b) # pipeline gets 21.6.
```

The result is written to the pipeline, as the top-level expression has no side effects.

7.1.2 Member access

Syntax:

Syntax

```
member-access:  
    primary-expression . new-line-opt~ member-name  
    primary-expression :: new-line-opt~ member-name
```

Note that no whitespace is allowed after *primary-expression*.

Description:

The operator `.` is used to select an instance member from an object, or a key from a `Hashtable`. The left operand must designate an object, and the right operand must designate an accessible instance member.

Either the right operand designates an accessible instance member within the type of the object designated by the left operand or, if the left operand designates an array, the right operand designates accessible instance members within each element of the array.

Whitespace is not permitted before the `.` operator.

This operator is left associative.

The operator `::` is used to select a static member from a given type. The left operand must designate a type, and the right-hand operand must designate an accessible static member within that type.

Whitespace is not permitted before the `::` operator.

This operator is left associative.

If the right-hand operand designates a writable location within the type of the object designated by the left operand, then the whole expression designates a writable location.

Examples:

PowerShell

```
$a = 10, 20, 30
$a.Length                                # get instance property

(10, 20, 30).Length

$property = "Length"
$a.$property                               # property name is a variable

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123
}
$h1.FirstName                            # designates the key FirstName
$h1.Keys                                  # gets the collection of keys

[int]::MinValue                         # get static property
[double]::PositiveInfinity             # get static property
$property = "MinValue"
[long]::$property                       # property name is a variable

foreach ($t in [byte], [int], [long]) {
    $t::.MaxValue                        # get static property
}

$a = @{ID = 1 }, @{ID = 2 }, @{ID = 3 }
$a.ID                                    # get ID from each element in the array
```

7.1.3 Invocation expressions

Syntax:

Syntax

```
invocation-expression:
    primary-expression . new-line~opt~ member-name argument-list
    primary-expression :: new-line~opt~ member-name argument-list

argument-list:
    ( argument-expression-list~opt~ new-lines~opt~ )
```

Note that no whitespace is allowed after *primary-expression*.

Description:

An *invocation-expression* calls the method designated by `primary-expression.member-name` or `primary-expression::member-name`. The parentheses in *argument-list* contain a possibly empty, comma-separated list of expressions that designate the *arguments* whose values are passed to the method. Before the method is called, the arguments are evaluated and converted according to the rules of §6, if necessary, to match the types expected by the method. The order of evaluation of `primary-expression.member-name`, `primary-expression::member-name`, and the arguments is unspecified.

This operator is left associative.

The type of the result of an *invocation-expression* is a *method-designator* (§4.5.24).

Examples:

PowerShell

```
[Math]::Sqrt(2.0)          # call method with argument 2.0
[char]::IsUpper("a")       # call method
$b = "abc#$%XYZabc"
$b.ToUpper()               # call instance method

[Math]::Sqrt(2)            # convert 2 to 2.0 and call method
[Math]::Sqrt(2D)           # convert 2D to 2.0 and call method
[Math]::Sqrt($true)         # convert $true to 1.0 and call method
[Math]::Sqrt("20")          # convert "20" to 20 and call method

$a = [Math]::Sqrt           # get method descriptor for Sqrt
$a.Invoke(2.0)              # call Sqrt via the descriptor
$a = [Math]::("Sq"+"rt")    # get method descriptor for Sqrt
$a.Invoke(2.0)              # call Sqrt via the descriptor
$a = [char]::ToLower        # get method descriptor for ToLower
$a.Invoke("X")               # call ToLower via the descriptor
```

7.1.4 Element access

Syntax:

Syntax

```
element-access:
    primary-expression [ new-lines~opt~ expression new-lines~opt~ ]
```

Description:

There must not be any whitespace between *primary-expression* and the left square bracket ([).

7.1.4.1 Subscripting an array

Description:

Arrays are discussed in detail in §9. If *expression* is a 1-dimensional array, see §7.1.4.5.

When *primary-expression* designates a 1-dimensional array *A*, the operator [] returns the element located at *A[0 + expression]* after the value of *expression* has been converted to *int*. The result has the element type of the array being subscripted. If *expression* is negative, *A[expression]* designates the element located at *A[A.Length + expression]*.

When *primary-expression* designates a 2-dimensional array *B*, the operator [] returns the element located at *B[0 + row, 0 + column]* after the value of the *row* and *column* components of *expression* (which are specified as a comma-separated list) have been converted to *int*. The result has the element type of the array being subscripted. Unlike for a 1-dimensional array, negative positions have no special meaning.

When *primary-expression* designates an array of three or more dimensions, the rules for 2-dimensional arrays apply and the dimension positions are specified as a comma-separated list of values.

If a read access on a non-existing element is attempted, the result is \$null. It is an error to write to a non-existing element.

For a multidimensional-array subscript expression, the order of evaluation of the dimension position expressions is unspecified. For example, given a 3-dimensional array \$a, the behavior of \$a[\$i++, \$i, ++\$i] is unspecified.

If *expression* is an array, see §7.1.4.5.

This operator is left associative.

Examples:

PowerShell

```
$a = [int[]](10,20,30) # [int[]], Length 3
$a[1] # returns int 20
$a[20] # no such position, returns $null
$a[-1] # returns int 30, i.e., $a[$a.Length-1]
$a[2] = 5 # changes int 30 to int 5
```

```

$a[20] = 5 # implementation-defined behavior

$a = New-Object 'double[,]' 3,2
$a[0,0] = 10.5 # changes 0.0 to 10.5
$a[0,0]++ # changes 10.5 to 10.6

$list = ("red",$true,10),20,(1.2, "yes")
$list[2][1] # returns string "yes"

$a = @{ A = 10 },@{ B = $true },@{ C = 123.45 }
$a[1]["B"] # $a[1] is a Hashtable, where B is a key

$a = "red","green"
$a[1][4] # returns string "n" from string in $a[1]

```

If a write access to a non-existing element is attempted, an `IndexOutOfRangeException` exception is raised.

7.1.4.2 Subscripting a string

Description:

When *primary-expression* designates a string *S*, the operator `[]` returns the character located in the zero-based position indicated by *expression*, as a `char`. If *expression* is greater than or equal to that string's length, the result is `$null`. If *expression* is negative, `S[expression]` designates the element located at `S[S.Length + expression]`.

Examples:

PowerShell

```

$s = "Hello"    # string, Length 5, positions 0-4
$c = $s[1]      # returns "e" as a string
$c = $s[20]     # no such position, returns $null
$c = $s[-1]     # returns "o", i.e., $s[$s.Length-1]

```

7.1.4.3 Subscripting a Hashtable

Description:

When *primary-expression* designates a Hashtable, the operator `[]` returns the value(s) associated with the key(s) designated by *expression*. The type of *expression* is not restricted.

When *expression* is a single key name, the result is the associated value and has that type, unless no such key exists, in which case, the result is `$null`. If `$null` is used as the

key the behavior is implementation defined. If *expression* is an array of key names, see §7.1.4.5.

If *expression* is an array, see §7.1.4.5.

Examples:

PowerShell

```
$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }
$h1['FirstName']      # the value associated with key FirstName
$h1['BirthDate']      # no such key, returns $null

$h1 = @{ 10 = "James"; 20.5 = "Anderson"; $true = 123 }
$h1[10]                # returns value "James" using key 10
$h1[20.5]              # returns value "Anderson" using key 20.5
$h1[$true]              # returns value 123 using key $true
```

When *expression* is a single key name, if `$null` is used as the only value to subscript a Hashtable, a `NullArrayIndex` exception is raised.

7.1.4.4 Subscripting an XML document

Description:

When *primary-expression* designates an object of type `xml`, *expression* is converted to string, if necessary, and the operator `[]` returns the first child element having the name specified by *expression*. The type of *expression* must be string. The type of the result is implementation defined. The result can be subscripted to return its first child element. If no child element exists with the name specified by *expression*, the result is `$null`. The result does not designate a writable location.

Examples:

PowerShell

```
$x = [xml]@"
<Name>
<FirstName>Mary</FirstName>
<LastName>King</LastName>
</Name>
"@

$x['Name']           # refers to the element Name
$x['Name']['FirstName'] # refers to the element FirstName within Name
$x['FirstName']       # No such child element at the top level, result
is $null
```

The type of the result is `System.Xml.XmlElement` or `System.String`.

7.1.4.5 Generating array slices

When *primary-expression* designates an object of a type that is enumerable (§4) or a `Hashtable`, and *expression* is a 1-dimensional array, the result is an array slice (§9.9) containing the elements of *primary-expression* designated by the elements of *expression*.

In the case of a `Hashtable`, the array slice contains the associated values to the keys provided, unless no such key exists, in which case, the corresponding element is `$null`.

If `$null` is used as any key name the behavior is implementation defined.

Examples:

PowerShell

```
$a = [int[]](30,40,50,60,70,80,90)
$a[1,3,5]                      # slice has Length 3, value 40,60,80
$a[,5]                          # slice with Length 1
$a[@()]                         # slice with Length 0
$a[-1..-3]                      # slice with Length 3, value 90,80,70
$a = New-Object 'int[,]' 3,2
$a[0,0] = 10; $a[0,1] = 20; $a[1,0] = 30
$a[1,1] = 40; $a[2,0] = 50; $a[2,1] = 60
$a[(0,1),(1,0)]                # slice with Length 2, value 20,30, parens needed
$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }
$h1['FirstName']                # the value associated with key FirstName
$h1['BirthDate']                # no such key, returns $null
$h1['FirstName','IDNum']        # returns [Object[]], Length 2 (James/123)
$h1['FirstName','xxx']          # returns [Object[]], Length 2 (James/$null)
$h1[$null,'IDNum']              # returns [Object[]], Length 2 ($null/123)
```

Windows PowerShell: When *expression* is a collection of two or more key names, if `$null` is used as any key name that key is ignored and has no corresponding element in the resulting array.

7.1.5 Postfix increment and decrement operators

Syntax:

Syntax

```
post-increment-expression:
    primary-expression ++
```

```
post-decrement-expression:  
    primary-expression dashdash
```

Description:

The *primary-expression* must designate a writable location having a value of numeric type (§4) or the value `$null`. If the value designated by the operand is `$null`, that value is converted to type int and value zero before the operator is evaluated. The type of the value designated by *primary-expression* may change when the result is stored. See §7.11 for a discussion of type change via assignment.

The result produced by the postfix `++` operator is the value designated by the operand. After that result is obtained, the value designated by the operand is incremented by 1 of the appropriate type. The type of the result of expression `E++` is the same as for the result of the expression `E + 1` (§7.7).

The result produced by the postfix `--` operator is the value designated by the operand. After that result is obtained, the value designated by the operand is decremented by 1 of the appropriate type. The type of the result of expression `E--` is the same as for the result of the expression `E - 1` (§7.7).

These operators are left associative.

Examples:

```
PowerShell  
  
$i = 0          # $i = 0  
$i++           # $i is incremented by 1  
$j = $i--       # $j takes on the value of $i before the decrement  
  
$a = 1,2,3  
$b = 9,8,7  
$i = 0  
$j = 1  
$b[$j--] = $a[$i++] # $b[1] takes on the value of $a[0], then $j is  
# decremented, $i incremented  
  
$i = 2147483647 # $i holds a value of type int  
$i++           # $i now holds a value of type double because  
# 2147483648 is too big to fit in type int  
  
[int]$k = 0      # $k is constrained to int  
$k = [int]::MaxValue # $k is set to 2147483647  
$k++           # 2147483648 is too big to fit, imp-def behavior
```

```
$x = $null          # target is unconstrained, $null goes to [int]0
$x++                # value treated as int, 0->1
```

7.1.6 \$(...) operator

Syntax:

Syntax

sub-expression:

```
$( new-lines~opt~ statement-list~opt~ new-lines~opt~ )
```

Description:

If *statement-list* is omitted, the result is `$null`. Otherwise, *statement-list* is evaluated. Any objects written to the pipeline as part of the evaluation are collected in an unconstrained 1-dimensional array, in order. If the array of collected objects is empty, the result is `$null`. If the array of collected objects contains a single element, the result is that element; otherwise, the result is the unconstrained 1-dimensional array of collected results.

Examples:

PowerShell

```
$j = 20
 $($i = 10) # pipeline gets nothing
 $($i = 10) # pipeline gets int 10
 $($i = 10; $j) # pipeline gets int 20
 $($i = 10; $j) # pipeline gets [Object[]](10,20)
 $($i = 10; ++$j) # pipeline gets int 10
 $($i = 10; (++$j)) # pipeline gets [Object[]](10,22)
 $($i = 10; ++$j) # pipeline gets nothing
 $(2,4,6) # pipeline gets [Object[]](2,4,6)
```

7.1.7 @(...) operator

Syntax:

Syntax

array-expression:

```
@( new-lines~opt~ statement-list~opt~ new-lines~opt~ )
```

Description:

If *statement-list* is omitted, the result is an unconstrained 1-dimensional array of length zero. Otherwise, *statement-list* is evaluated, and any objects written to the pipeline as part of the evaluation are collected in an unconstrained 1-dimensional array, in order. The result is the (possibly empty) unconstrained 1-dimensional array.

Examples:

PowerShell

```
$j = 20
@($i = 10)           # 10 not written to pipeline, result is array of 0
@(($i = 10))         # pipeline gets 10, result is array of 1
@($i = 10; $j)        # 10 not written to pipeline, result is array of 1
@(($i = 10); $j)      # pipeline gets 10, result is array of 2
@(($i = 10); ++$j)    # pipeline gets 10, result is array of 1
@(($i = 10); (++$j))  # pipeline gets both values, result is array of 2
@($i = 10; ++$j)       # pipeline gets nothing, result is array of 0

$a = @(2,4,6)          # result is array of 3
@($a)                  # result is the same array of 3
(@(@($a)))             # result is the same array of 3
```

7.1.8 Script block expression

Syntax:

Syntax

```
script-block-expression:
{ new-lines~opt~ script-block new-lines~opt~ }

script-block:
param-block~opt~ statement-terminators~opt~ script-block-body~opt~

script-block-body:
named-block-list
statement-list
```

Description:

param-block is described in §8.10.9. *named-block-list* is described in §8.10.7.

A script block is an unnamed block of statements that can be used as a single unit. Script blocks can be used to invoke a block of code as if it was a single command, or they can be assigned to variables that can be executed.

The *named-block-list* or *statement-list* is executed and the type and value(s) of the result are the type and value(s) of the results of those statement sets.

A *script-block-expression* has type `scriptblock` ([§4.3.7](#)).

If *param-block* is omitted, any arguments passed to the script block are available via `$args` ([§8.10.1](#)).

During parameter binding, a script block can be passed either as a script block object or as the result after the script block has been evaluated. See [§6.17](#) for further information.

7.1.9 Hash literal expression

Syntax:

```
Syntax

hash-literal-expression:
    @{ new-lines~opt~ hash-literal-body~opt~ new-lines~opt~ }

hash-literal-body:
    hash-entry
    hash-literal-body statement-terminators hash-entry

hash-entry:
    key-expression = new-lines~opt~ statement

key-expression:
    simple-name
    unary-expression

statement-terminators:
    statement-terminator
    statement-terminators statement-terminator

statement-terminator:
    ;
    new-line-character
```

Description:

A *hash-literal-expression* is used to create a Hashtable ([§10](#)) of zero or more elements each of which is a key/value pair.

The key may have any type except the null type. The associated values may have any type, including the null type, and each of those values may be any expression that designates the desired value, including `$null`.

The ordering of the key/value pairs is not significant.

Examples:

PowerShell

```
$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }
$last = "Anderson"; $IDNum = 120
$h2 = @{ FirstName = "James"; LastName = $last; IDNum = $IDNum + 3 }
$h3 = @{
$h4 = @{ 10 = "James"; 20.5 = "Anderson"; $true = 123 }
```

which creates two Hashtables, `$h1` and `$h2`, each containing three key/value pairs, and a third, `$h3`, that is empty. Hashtable `$h4` has keys of various types.

7.1.10 Type literal expression

Syntax:

Syntax

```
type-literal:
    [ type-spec ]

type-spec:
    array-type-name new-lines~opt~ dimension~opt~ ]
    generic-type-name new-lines~opt~ generic-type-arguments ]
    type-name

dimension:
    ,
    dimension ,

generic-type-arguments:
    type-spec new-lines~opt~
    generic-type-arguments , new-lines~opt~ type-spec

array-type-name:
    type-name []

generic-type-name:
    type-name []
```

Description:

A *type-literal* is represented in an implementation by some unspecified *underlying type*. As a result, a type name is a synonym for its underlying type.

Type literals are used in a number of contexts:

- Specifying an explicit conversion (§6, [§7.2.9](#))
- Creating a type-constrained array ([§9.4](#))
- Accessing the static members of an object ([§7.1.2](#))
- Specifying a type constraint on a variable ([§5.3](#)) or a function parameter ([§8.10.2](#))

Examples:

PowerShell

```
[int].IsPrimitive      # $true
[Object[]].FullName    # "System.Object[]"
[int[,]].GetArrayRank() # 3
```

A generic stack type ([§4.4](#)) that is specialized to hold strings might be written as `[Stack[string]]`, and a generic dictionary type that is specialized to hold `int` keys with associated string values might be written as `[Dictionary[int,string]]`.

The type of a *type-literal* is `System.Type`. The complete name for the type `Stack[string]` suggested above is `System.Collections.Generic.Stack[int]`. The complete name for the type `Dictionary[int,string]` suggested above is `System.Collections.Generic.Dictionary[int,string]`.

7.2 Unary operators

Syntax:

Syntax

```
unary-expression:
    primary-expression
    expression-with-unary-operator

expression-with-unary-operator:
    , new-lines~opt~ unary-expression
    -not new-lines~opt~ unary-expression
    ! new-lines~opt~ unary-expression
    -bnot new-lines~opt~ unary-expression
    + new-lines~opt~ unary-expression
    dash new-lines~opt~ unary-expression
    pre-increment-expression
    pre-decrement-expression
    cast-expression
    -split new-lines~opt~ unary-expression
    -join new-lines~opt~ unary-expression
```

```

pre-increment-expression:
++ new-lines~opt~ unary-expression

pre-decrement-expression:
dashdash new-lines~opt~ unary-expression

dash: one of
- (U+002D)
EnDash character (U+2013)
EmDash character (U+2014)
Horizontal bar character (U+2015)

cast-expression:
type-literal unary-expression

dashdash:
dash dash

```

7.2.1 Unary comma operator

Description:

The comma operator (,) creates an unconstrained 1-dimensional array having one element, whose type and value are that of *unary-expression*.

This operator is right associative.

Examples:

PowerShell

```

$a = ,10          # create an unconstrained array of 1 element, $a[0],
# which has type int

$a = ,(10,"red") # create an unconstrained array of 1 element,
$a[0],           # which is an unconstrained array of 2 elements,
# $a[0][0] an int, and $a[0][1] a string

$a = ,,10         # create an unconstrained array of 1 element, which is
# an unconstrained array of 1 element, which is an int
# $a[0][0] is the int. Contrast this with @(@(10))

```

7.2.2 Logical NOT

Syntax:

Syntax

```
logical-not-operator:  
    dash not  
  
dash: one of  
    - (U+002D)  
        EnDash character (U+2013)  
        EmDash character (U+2014)  
        Horizontal bar character (U+2015)
```

Description:

The operator `-not` converts the value designated by *unary-expression* to type `bool` ([§6.2](#)), if necessary, and produces a result of that type. If *unary-expression*'s value is `True`, the result is `False`, and vice versa. The operator `!` is an alternate spelling for `-not`.

This operator is right associative.

Examples:

PowerShell

```
-not $true          # False  
-not -not $false   # False  
-not 0             # True  
-not 1.23          # False  
!"xyz"            # False
```

7.2.3 Bitwise NOT

Syntax:

Syntax

```
bitwise-not-operator:  
    dash bnot  
  
dash: one of  
    - (U+002D)  
        EnDash character (U+2013)  
        EmDash character (U+2014)  
        Horizontal bar character (U+2015)
```

Description:

The operator `-bnot` converts the value designated by *unary-expression* to an integer type ([§6.4](#)), if necessary. If the converted value can be represented in type `int` then that is

the result type. Else, if the converted value can be represented in type `long` then that is the result type. Otherwise, the expression is ill-formed. The resulting value is the ones-complement of the converted value.

This operator is right associative.

Examples:

PowerShell

```
-bnot $true          # int with value 0xFFFFFFFF  
-bnot 10            # int with value 0xFFFFFFFF5  
-bnot 2147483648.1 # long with value 0xFFFFFFFF7FFFFFFF  
-bnot $null          # int with value 0xFFFFFFFF  
-bnot "0abc"         # int with value 0xFFFFF543
```

7.2.4 Unary plus

Description:

An expression of the form `+ unary-expression` is treated as if it were written as `0 + unary-expression` ([§7.7](#)). The integer literal `0` has type `int`.

This operator is right associative.

Examples:

PowerShell

```
+123L      # type long, value 123  
+0.12340D  # type decimal, value 0.12340  
+"0abc"    # type int, value 2748
```

7.2.5 Unary minus

Description:

An expression of the form `- unary-expression` is treated as if it were written as `0 - unary-expression` ([§7.7](#)). The integer literal 0 has type `int`. The minus operator can be any one of the *dash* characters listed in [§7.2](#).

This operator is right associative.

Examples:

PowerShell

```
-$true      # type int, value -1
-123L      # type long, value -123
-0.12340D # type decimal, value -0.12340
```

7.2.6 Prefix increment and decrement operators

Description:

The *unary-expression* must designate a writable location having a value of numeric type ([§4](#)) or the value `$null`. If the value designated by its *unary-expression* is `$null`, *unary-expression*'s value is converted to type int and value zero before the operator is evaluated.

ⓘ Note

The type of the value designated by *unary-expression* may change when the result is stored. See [§7.11](#) for a discussion of type change via assignment.

For the prefix increment operator `++`, the value of *unary-expression* is incremented by `1` of the appropriate type. The result is the new value after incrementing has taken place. The expression `++E` is equivalent to `E += 1` ([§7.11.2](#)).

For the prefix decrement operator `--`, the value of *unary-expression* is decremented by `1` of the appropriate type. The result is the new value after decrementing has taken place. The expression `--E` is equivalent to `E -= 1` ([§7.11.2](#)). The prefix decrement operator can be any of the patterns matching the *dashdash* pattern in [§7.2](#).

These operators are right associative.

Examples:

PowerShell

```
$i = 0          # $i = 0
++$i           # $i is incremented by 1
$j = --$i       # $i is decremented then $j takes on the value of $i

$a = 1,2,3
$b = 9,8,7
$i = 0;
$j = 1
$b[--$j] = $a[++$i] # $j is # decremented, $i incremented, then $b[0]
                     # takes on the value of $a[1]
```

```

$! = 2147483647      # $! holds a value of type int
+$!
# $! now holds a value of type double because
# 2147483648 is too big to fit in type int

[int]$k = 0            # $k is constrained to int
$k = [int]::MinValue  # $k is set to -2147483648
--$k                  # -2147483649 is too small to fit, imp-def behavior

$x = $null             # target is unconstrained, $null goes to [int]0
--$x                  # value treated as int, 0 becomes -1

```

7.2.7 The unary -join operator

Syntax:

Syntax
<pre> join-operator: dash join dash: one of - (U+002D) EnDash character (U+2013) EmDash character (U+2014) Horizontal bar character (U+2015) </pre>

Description:

The unary `-join` operator produces a string that is the concatenation of the value of one or more objects designated by *unary-expression*. (A separator can be inserted by using the binary version of this operator ([§7.8.4.4](#))).

unary-expression can be a scalar value or a collection.

Examples:

PowerShell
<pre> -join (10, 20, 30) # result is "102030" -join (123, \$false, 19.34e17) # result is "123False1.934E+18" -join 12345 # result is "12345" -join \$null # result is "" </pre>

7.2.8 The unary -split operator

Syntax:

Syntax

```
split-operator:  
  dash split  
  
dash: one of  
  - (U+002D)  
    EnDash character (U+2013)  
    EmDash character (U+2014)  
    Horizontal bar character (U+2015)
```

Description:

The unary `-split` operator splits one or more strings designated by *unary-expression*, returning their subparts in a constrained 1-dimensional array of string. It treats any contiguous group of whitespace characters as the delimiter between successive subparts. An explicit delimiter string can be specified by using the binary version of this operator ([§7.8.4.5](#)) or its two variants ([§7.8](#)).

The delimiter text is not included in the resulting strings. Leading and trailing whitespace in the input string is ignored. An input string that is empty or contains whitespace only results in an array of one string, which is empty.

unary-expression can designate a scalar value or an array of strings.

Examples:

PowerShell

```
-split " red`tblue`ngreen " # 3 strings: "red", "blue", "green"  
-split ("yes no", "up down") # 4 strings: "yes", "no", "up", "down"  
-split " " # 1 (empty) string
```

7.2.9 Cast operator

Description:

This operator converts explicitly ([§6](#)) the value designated by *unary-expression* to the type designated by *type-literal* ([§7.1.10](#)). If *type-literal* is other than void, the type of the result is the named type, and the value is the value after conversion. If *type-literal* is void, no object is written to the pipeline and there is no result.

When an expression of any type is cast to that same type, the resulting type and value is the *unary-expression*'s type and value.

This operator is right associative.

Examples:

PowerShell

```
[bool]-10      # a bool with value True
[int]-10.70D   # a decimal with value -10
[int]10.7      # an int with value 11
[long]"+2.3e+3" # a long with value 2300
[char[]]"Hello" # an array of 5 char with values H, e, l, l, and o.
```

7.3 Binary comma operator

Syntax:

Syntax

```
array-literal-expression:
    unary-expression , new-lines~opt~ array-literal-expression
```

Description:

The binary comma operator creates a 1-dimensional array whose elements are the values designated by its operands, in lexical order. The array has unconstrained type.

Examples:

PowerShell

```
2,4,6          # Length 3; values 2,4,6
(2,4),6        # Length 2; values [Object[],int]
(2,4,6),12,(2..4) # Length 3; [Object[],int,[Object[]]]
2,4,6,"red",$null,$true # Length 6
```

The addition of grouping parentheses to certain binary comma expressions does not document the default precedence; instead, it changes the result.

7.4 Range operator

Syntax:

Syntax

```
range-expression:  
    unary-expression .. new-lines~opt~ unary-expression
```

Description:

A *range-expression* creates an unconstrained 1-dimensional array whose elements are the values of the `int` sequence specified by the range bounds. The values designated by the operands are converted to `int`, if necessary ([§6.4](#)). The operand designating the lower value after conversion is the *lower bound*, while the operand designating the higher value after conversion is the *upper bound*. Both bounds may be the same, in which case, the resulting array has length 1. If the left operand designates the lower bound, the sequence is in ascending order. If the left operand designates the upper bound, the sequence is in descending order.

Conceptually, this operator is a shortcut for the corresponding binary comma operator sequence. For example, the range `5..8` can also be generated using `5,6,7,8`. However, if an ascending or descending sequence is needed without having an array, an implementation may avoid generating an actual array. For example, in `foreach ($i in 1..5) { ... }`, no array need be created.

A *range-expression* can be used to specify an array slice ([§9.9](#)).

Examples:

PowerShell

```
1..10      # ascending range 1..10  
-495..-500 # descending range -495..-500  
16..16     # sequence of 1  
  
$x = 1.5  
$x..5.40D # ascending range 2..5  
  
$true..3   # ascending range 1..3  
-2..$null  # ascending range -2..0  
0xf..0xa   # descending range 15..10
```

7.5 Format operator

Syntax:

Syntax

```

format-expression:
    format-specification-string format-operator new-lines~opt~ range-
expression

format-operator:
    dash f

dash:
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)

```

Description:

A *format-expression* formats one or more values designated by *range-expression* according to a *format-specification-string* designated by *format-expression*. The positions of the values designated by *range-expression* are numbered starting at zero and increasing in lexical order. The result has type `string`.

A format specification string may contain zero or more format specifications each having the following form:

```
{N [ ,M ][ : FormatString ]}
```

N represents a (required) *range-expression* value position, *M* represents the (optional) minimum display width, and *FormatString* indicates the (optional) format. If the width of a formatted value exceeds the specified width, the width is increased accordingly. Values whose positions are not referenced in *FormatString* are ignored after being evaluated for any side effects. If *N* refers to a non-existent position, the behavior is implementation defined. Value of type `$null` and `void` are formatted as empty strings. Arrays are formatted as for *sub-expression* (§7.1.6). To include the characters `{` and `}` in a format specification without their being interpreted as format delimiters, write them as `{}{` and `}}{`, respectively.

For a complete definition of format specifications, see the type `System.IFormattable` in [Ecma Technical Report TR/84](#).

Examples:

PowerShell

<code>"__{0,3}__" -f 5</code>	<code># __ 5__</code>
<code>"__{0,-3}__" -f 5</code>	<code># __5__</code>
<code>"__{0,3:000}__" -f 5</code>	<code># __005__</code>
<code>"__{0,5:0.00}__" -f 5.0</code>	<code># __ 5.00__</code>

```

"__{0:C}__" -f 1234567.888          # __$1,234,567.89__
"__{0:C}__" -f -1234.56             # __($1,234.56)__
"__{0,12:e2}__" -f 123.456e2        # __ 1.23e+004__
"__{0,-12:p}__" -f -0.252           # __-25.20 % __

$i = 5; $j = 3
"__{0} + {1} <= {2}__" -f $i,$j,($i+$j) # __5 + 3 <= 8__

$format = "__0x{0:X8}__"
$format -f 65535                      # __0x0000FFFF__

```

In a format specification, if N refers to a non-existent position, a `FormatError` is raised.

7.6 Multiplicative operators

Syntax:

Syntax

```

multiplicative-expression:
    multiplicative-expression * new-lines~opt~ format-expression
    multiplicative-expression / new-lines~opt~ format-expression
    multiplicative-expression % new-lines~opt~ format-expression

```

7.6.1 Multiplication

Description:

The result of the multiplication operator `*` is the product of the values designated by the two operands after the usual arithmetic conversions ([§6.15](#)) have been applied.

This operator is left associative.

Examples:

PowerShell

```

12 * -10L      # long result -120
-10.300D * 12 # decimal result -123.600
10.6 * 12     # double result 127.2
12 * "0xabc"   # int result 32976

```

7.6.2 String replication

Description:

When the left operand designates a string the binary `*` operator creates a new string that contains the one designated by the left operand replicated the number of times designated by the value of the right operand as converted to integer type ([§6.4](#)).

This operator is left associative.

Examples:

PowerShell

```
"red" * "3"      # string replicated 3 times
"red" * 4        # string replicated 4 times
"red" * 0        # results in an empty string
"red" * 2.3450D  # string replicated twice
"red" * 2.7      # string replicated 3 times
```

7.6.3 Array replication

Description:

When the left operand designates an array the binary `*` operator creates a new unconstrained 1-dimensional array that contains the value designated by the left operand replicated the number of times designated by the value of the right operand as converted to integer type ([§6.4](#)). A replication count of zero results in an array of length 1. If the left operand designates a multidimensional array, it is flattened ([§9.12](#)) before being used.

This operator is left associative.

Examples:

PowerShell

```
$a = [int[]](10,20)          # [int[]], Length 2*1
$a * "3"                    # [Object[]], Length 2*3
$a * 4                      # [Object[]], Length 2*4
$a * 0                      # [Object[]], Length 2*0
$a * 2.3450D                # [Object[]], Length 2*2
$a * 2.7                    # [Object[]], Length 2*3
(New-Object 'float[,]' 2,3) * 2 # [Object[]], Length 2*2
```

7.6.4 Division

Description:

The result of the division operator `/` is the quotient when the value designated by the left operand is divided by the value designated by the right operand after the usual arithmetic conversions ([§6.15](#)) have been applied.

If an attempt is made to perform integer or decimal division by zero, an implementation-defined terminating error is raised.

This operator is left associative.

Examples:

PowerShell

```
10/-10      # int result -1
12/-10      # double result -1.2
12/-10D     # decimal result 1.2
12/10.6     # double result 1.13207547169811
12/"0xabc"  # double result 0.00436681222707424
```

If an attempt is made to perform integer or decimal division by zero, a **RuntimeException** exception is raised.

7.6.5 Remainder

Description:

The result of the remainder operator `%` is the remainder when the value designated by the left operand is divided by the value designated by the right operand after the usual arithmetic conversions ([§6.15](#)) have been applied.

If an attempt is made to perform integer or decimal division by zero, an implementation-defined terminating error is raised.

Examples:

PowerShell

```
10 % 3      # int result 1
10.0 % 0.3   # double result 0.1
10.00D % "0x4" # decimal result 2.00
```

If an attempt is made to perform integer or decimal division by zero, a **RuntimeException** exception is raised.

7.7 Additive operators

Syntax:

```
Syntax

additive-expression:
    primary-expression + new-lines~opt~ expression
    primary-expression dash new-lines~opt~ expression

dash: one of
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)
```

7.7.1 Addition

Description:

The result of the addition operator `+` is the sum of the values designated by the two operands after the usual arithmetic conversions ([§6.15](#)) have been applied.

This operator is left associative.

Examples:

```
PowerShell

12 + -10L      # long result 2
-10.300D + 12  # decimal result 1.700
10.6 + 12      # double result 22.6
12 + "0xabc"   # int result 2760
```

7.7.2 String concatenation

Description:

When the left operand designates a string the binary `+` operator creates a new string that contains the value designated by the left operand followed immediately by the value(s) designated by the right operand as converted to type string ([§6.8](#)).

This operator is left associative.

Examples:

PowerShell

```
"red" + "blue"      # "redblue"  
"red" + "123"       # "red123"  
"red" + 123         # "red123"  
"red" + 123.456e+5 # "red12345600"  
"red" + (20,30,40)  # "red20 30 40"
```

7.7.3 Array concatenation

Description:

When the left operand designates an array the binary `+` operator creates a new unconstrained 1-dimensional array that contains the elements designated by the left operand followed immediately by the value(s) designated by the right operand. Multidimensional arrays present in either operand are flattened ([§9.12](#)) before being used.

This operator is left associative.

Examples:

PowerShell

```
$a = [int[]](10,20)                  # [int[]], Length 2  
$a + "red"                          # [Object[]], Length 3  
$a + 12.5,$true                     # [Object[]], Length 4  
$a + (New-Object 'float[,]' 2,3)    # [Object[]], Length 8  
(New-Object 'float[,]' 2,3) + $a    # [Object[]], Length 8
```

7.7.4 Hashtable concatenation

Description:

When both operands designate Hashtables the binary `+` operator creates a new Hashtable that contains the elements designated by the left operand followed immediately by the elements designated by the right operand.

If the Hashtables contain the same key, an implementation-defined terminating error is raised.

This operator is left associative.

Examples:

PowerShell

```
$h1 = @{ FirstName = "James"; LastName = "Anderson" }
$h2 = @{ Dept = "Personnel" }
$h3 = $h1 + $h2      # new Hashtable, Count = 3
```

If the Hashtables contain the same key, an exception of type **BadOperatorArgument** is raised.

7.7.5 Subtraction

Description:

The result of the subtraction operator `-` is the difference when the value designated by the right operand is subtracted from the value designated by the left operand after the usual arithmetic conversions ([§6.15](#)) have been applied. The subtraction operator can be any one of the *dash* characters listed in [§7.7](#).

This operator is left associative.

Examples:

PowerShell

```
12 - -10L      # long result 22
-10.300D - 12 # decimal result -22.300
10.6 - 12        # double result -1.4
12 - "0xabC"     # int result -2736
```

7.8 Comparison operators

Syntax:

Syntax

```
comparison-expression:
    primary-expression comparison-operator new-lines~opt~ expression

comparison-operator:
    equality-operator
    relational-operator
    containment-operator
    type-operator
    like-operator
    match-operator
```

Description:

The type of the value designated by the left operand determines how the value designated by the right operand is converted ([§6](#)), if necessary, before the comparison is done.

Some comparison operators have two variants, one that is case sensitive (`-c<operator>`), and one that isn't case sensitive (`-i<operator>`). The `-<operator>` version is equivalent to `-i<operator>`. Case sensitivity is meaningful only with comparisons of values of type string. In non-string comparison contexts, the two variants behave the same.

These operators are left associative.

7.8.1 Equality and relational operators

Syntax:

```
Syntax

equality-operator: one of
    dash eq      dash ceq      dash ieq
    dash ne      dash cne      dash ine

dash:
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)

relational-operator: one of
    dash lt      dash clt     dash ilt
    dash le      dash cle     dash ile
    dash gt      dash cgt     dash igt
    dash ge      dash cge     dash ige
```

Description:

There are two equality operators: equality (`-eq`) and inequality (`-ne`); and four relational operators: less-than (`-lt`), less-than-or-equal-to (`-le`), greater-than (`-gt`), and greater-than-or-equal-to (`-ge`). Each of these has two variants ([§7.8](#)).

For two strings to compare equal, they must have the same length and contents, and letter case, if appropriate.

If the value designated by the left operand is not a collection, the result has type `bool`. Otherwise, the result is a possibly empty unconstrained 1-dimensional array containing

the elements of the collection that test True when compared to the value designated by the right operand.

Examples:

PowerShell

```
10 -eq "010"          # True, int comparison
"010" -eq 10          # False, string comparison
"RED" -eq "Red"        # True, case-insensitive comparison
"RED" -ceq "Red"       # False, case-sensitive comparison
"ab" -lt "abc"         # True

10,20,30,20,10 -ne 20 # 10,30,10, Length 3
10,20,30,20,10 -eq 40 # Length 0
10,20,30,20,10 -ne 40 # 10,20,30,20,10, Length 5
10,20,30,20,10 -gt 25 # 30, Length 1
0,1,30 -ne $true      # 0,30, Length 2
0,"00" -eq "0"         # 0 (int), Length 1
```

7.8.2 Containment operators

Syntax:

Syntax

```
containment-operator: one of
    dash contains      dash ccontains      dash icontains
    dash notcontains   dash cnotcontains   dash inotcontains
    dash in            dash cin           dash iin
    dash notin         dash cnotin        dash inotin

dash:
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)
```

Description:

There are four containment operators: contains (-contains), does-not-contain (-notcontains), in (-in) and not-in (-notin). Each of these has two variants ([§7.8](#)).

The containment operators return a result of type bool that indicates whether a value occurs (or does not occur) at least once in the elements of an array. With -contains and -notcontains, the value is designated by the right operand and the array is designated

by the left operand. With `-in` and `-notin`, the operands are reversed. The value is designated by the left operand and the array is designated by the right operand.

For the purposes of these operators, if the array operand has a scalar value, the scalar value is treated as an array of one element.

Examples:

PowerShell

```
10,20,30,20,10 -contains 20      # True
10,20,30,20,10 -contains 42.9    # False
10,20,30 -contains "10"          # True
"10",20,30 -contains 10          # True
"010",20,30 -contains 10          # False
10,20,30,20,10 -notcontains 15    # True
"Red",20,30 -ccontains "RED"      # False
```

7.8.3 Type testing and conversion operators

Syntax:

Syntax

```
type-operator: one of
  dash is
  dash as

dash:
  - (U+002D)
  EnDash character (U+2013)
  EmDash character (U+2014)
  Horizontal bar character (U+2015)
```

Description:

The type operator `-is` tests whether the value designated by the left operand has the type, or is derived from a type that has the type, designated by the right operand. The right operand must designate a type or a value that can be converted to a type (such as a string that names a type). The type of the result is `bool`. The type operator `-isnot` returns the logical negation of the corresponding `-is` form.

The type operator `-as` attempts to convert the value designated by the left operand to the type designated by the right operand. The right operand must designate a type or a value that can be converted to a type (such as a string that names a type). If the

conversion fails, `$null` is returned; otherwise, the converted value is returned and the return type of that result is the runtime type of the converted value.

Examples:

PowerShell

```
$a = 10          # value 10 has type int
$a -is [int]      # True

$t = [int]
$a -isnot $t      # False
$a -is "int"       # True
$a -isnot [double] # True

$x = [int[]](10,20)
$x -is [int[]}     # True

$a = "abcd"        # string is derived from object
$a -is [Object]    # True

$x = [double]
foreach ($t in [int],$x,[decimal],"string") {
    $b = (10.60D -as $t) * 2 # results in int 22, double 21.2
                                # decimal 21.20, and string "10.6010.60"
}
```

7.8.4 Pattern matching and text manipulation operators

7.8.4.1 The -like and -notlike operators

Syntax:

Syntax

```
like-operator: one of
    dash like      dash clike      dash ilike
    dash notlike   dash cnotlike   dash inotlike

dash:
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)
```

Description:

If the left operand does not designate a collection, the result has type `bool`. Otherwise, the result is a possibly empty unconstrained 1-dimensional array containing the

elements of the collection that test True when compared to the value designated by the right operand. The right operand may designate a string that contains wildcard expressions ([§3.15](#)). These operators have two variants ([§7.8](#)).

Examples:

PowerShell

```
"Hello" -like "h*"          # True, starts with h
"Hello" -clike "h*"         # False, does not start with lowercase
h
"Hello" -like "*1*"         # True, has an 1 in it somewhere
"Hello" -like "?1"          # False, no length match

"-abc" -like "[xz]*"        # True, - is not a range separator
#$%^&" -notlike "*[A-Za-z]" # True, does not end with alphabetic
character
"He" -like "h[aeiou]?"     # False, need at least 3 characters
"When" -like "*[?]"         # False, ? is not a wildcard character
"When?" -like "*[?]"        # True, ? is not a wildcard character

"abc","abbcde","abcgh" -like "abc*" # object[2], values
"abc" and "abcgh"
```

7.8.4.2 The `-match` and `-notmatch` operators

Syntax:

Syntax

```
match-operator: one of
    dash match      dash cmatch      dash imatch
    dash notmatch   dash cnotmatch   dash inotmatch

dash:
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)
```

Description:

If the left operand does not designate a collection, the result has type `bool` and if that result is `$true`, the elements of the Hashtable `$Matches` are set to the strings that match (or do-not-match) the value designated by the right operand. Otherwise, the result is a possibly empty unconstrained 1-dimensional array containing the elements of the collection that test True when compared to the value designated by the right operand,

and `$Matches` is not set. The right operand may designate a string that contains regular expressions ([§3.16](#)), in which case, it is referred to as a *pattern*. These operators have two variants ([§7.8](#)).

These operators support submatches ([§7.8.4.6](#)).

Examples:

PowerShell

```
"Hello" -match ".1"                      # True, $Matches key/value is 0/"el"
"Hello" -match '\^h.*o$'                   # True, $Matches key/value is
0/"Hello"
"Hello" -cmatch '\^h.*o$'                  # False, $Matches not set
"abc\^ef" -match ".\\|^e"                 # True, $Matches key/value is
0/"c\^e"

"abc" -notmatch "[A-Za-z]"                # False
"abc" -match "[\^A-Za-z]"                 # False
"He" -match "h[aeiou]."
"abc","abbcde","abcgh" -match "abc.*"    # Length is 2, values "abc", "abcgh"
```

7.8.4.3 The `-replace` operator

Syntax:

Syntax

```
binary-replace-operator: one of
    dash replace      dash creplace     dash ireplace

dash:
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)
```

Description:

The `-replace` operator allows text replacement in one or more strings designated by the left operand using the values designated by the right operand. This operator has two variants ([§7.8](#)). The right operand has one of the following forms:

- The string to be located, which may contain regular expressions ([§3.16](#)). In this case, the replacement string is implicitly "".
- An array of 2 objects containing the string to be located, followed by the replacement string.

If the left operand designates a string, the result has type string. If the left operand designates a 1-dimensional array of string, the result is an unconstrained 1-dimensional array, whose length is the same as for left operand's array, containing the input strings after replacement has completed.

This operator supports submatches ([§7.8.4.6](#)).

Examples:

PowerShell

```
"Analogous","an apple" -replace "a","*"      # "*n*logous","*n *pple"
"Analogous" -creplace "[aeiou]","?"          # An?l?g??s"
"Analogous","an apple" -replace '\^a','%A'    # %%Analogous",%%An apple"
"Analogous" -replace "[aeiou]','$&$&'       # AAnaaloogooous"
```

7.8.4.4 The binary `-join` operator

Syntax:

Syntax

```
binary-join-operator: one of
    dash join

dash:
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)
```

Description:

The binary `-join` operator produces a string that is the concatenation of the value of one or more objects designated by the left operand after having been converted to string ([§6.7](#)), if necessary. The string designated by the right operand is used to separate the (possibly empty) values in the resulting string.

The left operand can be a scalar value or a collection.

Examples:

PowerShell

```
(10, 20, 30) -join "\|"    # result is "10\|20\|30"
12345 -join ","            # result is "12345", no separator needed
```

```
($null,$null) -join "<->" # result is "<->", two zero-length values
```

7.8.4.5 The binary `-split` operator

Syntax:

Syntax

```
binary-split-operator: one of
    dash split      dash csplit     dash isplit

dash:
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)
```

Description:

The binary `-split` operator splits one or more strings designated by the left operand, returning their subparts in a constrained 1-dimensional array of string. This operator has two variants ([§7.8](#)). The left operand can designate a scalar value or an array of strings. The right operand has one of the following forms:

- A *delimiter string*
- An array of 2 objects containing a delimiter string followed by a numeric *split count*
- An array of 3 objects containing a delimiter string, a numeric split count, and an *options string*
- A script block
- An array of 2 objects containing a script block followed by a numeric split count

The delimiter string may contain regular expressions ([§3.16](#)). It is used to locate subparts with the input strings. The delimiter is not included in the resulting strings. If the left operand designates an empty string, that results in an empty string element. If the delimiter string is an empty string, it is found at every character position in the input strings.

By default, all subparts of the input strings are placed into the result as separate elements; however, the split count can be used to modify this behavior. If that count is negative, zero, or greater than or equal to the number of subparts in an input string, each subpart goes into a separate element. If that count is less than the number of subparts in the input string, there are count elements in the result, with the final element containing all of the subparts beyond the first **count - 1** subparts.

An options string contains zero or more *option names* with each adjacent pair separated by a comma. Leading, trailing, and embedded whitespace is ignored. Option names may be in any order and are case-sensitive.

If an options string contains the option name **SimpleMatch**, it may also contain the option name **IgnoreCase**. If an options string contains the option name **RegexMatch** or it does not contain either **RegexMatch** or **SimpleMatch**, it may contain any option name except **SimpleMatch**. However, it must not contain both **Multiline** and **Singleline**.

Here is the set of option names:

[+] Expand table

Option	Description
CultureInvariant	Ignores cultural differences in language when evaluating the delimiter.
ExplicitCapture	Ignores non-named match groups so that only explicit capture groups are returned in the result list.
IgnoreCase	Force case-insensitive matching, even if <code>-csplit</code> is used.
IgnorePatternWhitespace	Ignores unescaped whitespace and comments marked with the number sign (#).
Multiline	This mode recognizes the start and end of lines and strings. The default mode is Singleline .
RegexMatch	Use regular expression matching to evaluate the delimiter. This is the default.
SimpleMatch	Use simple string comparison when evaluating the delimiter.
Singleline	This mode recognizes only the start and end of strings. It is the default mode.

The script block ([§7.1.8](#)) specifies the rules for determining the delimiter, and must evaluate to type bool.

Examples:

PowerShell

```
"one,forty two," -split ","          # 5 strings: "one" "forty two" ""  
"  
"abc","de" -split ""                # 9 strings: "" "a" "b" "c" "" ""  
"d" "e" ""
```

```

"ab,cd","1,5,7,8" -split "," , 2           # 4 strings: "ab" "cd" "1" "5,7,8"

"10X20x30" -csplit "X", 0, "SimpleMatch"  # 2 strings: "10" "20x30"

"analogous" -split "[AEIOU]", 0, "RegexMatch, IgnoreCase"
                                         # 6 strings: "" "n" "l" "g" "" "s"

"analogous" -split { $_ -eq "a" -or $_ -eq "o" }, 4
                                         # 4 strings: "" "n" "l" "gous"

```

7.8.4.6 Submatches

The pattern being matched by `-match`, `-notmatch`, and `-replace` may contain subparts (called *submatches*) delimited by parentheses. Consider the following example:

```
"red" -match "red"
```

The result is `$true` and key 0 of `$Matches` contains "red", that part of the string designated by the left operand that exactly matched the pattern designated by the right operand.

In the following example, the whole pattern is a submatch:

```
"red" -match "(red)"
```

As before, key 0 contains "red"; however, key 1 also contains "red", which is that part of the string designated by the left operand that exactly matched the submatch.

Consider the following, more complex, pattern:

```
"red" -match "((r)e)(d)"
```

This pattern allows submatches of "re", "r", "d", or "red".

Again, key 0 contains "red". Key 1 contains "re", key 2 contains "r", and key 3 contains "d". The key/value pairs are in matching order from left-to-right in the pattern, with longer string matches preceding shorter ones.

In the case of `-replace`, the replacement text can access the submatches via names of the form `$n`, where the first match is `$1`, the second is `$3`, and so on. For example,

PowerShell

```
"Monday morning" -replace '(Monday|Tuesday)(morning|afternoon|evening)', 'the $2 of $1'
```

The resulting string is "the morning of Monday".

Instead of having keys in `$Matches` be zero-based indexes, submatches can be named using the form `?<name*>`. For example, `"((r)e)(d)"` can be written with three named submatches, `m1`, `m2`, and `m3`, as follows: `"(?<m1>(?<m2>r)e)(?<m3>d)"`.

7.8.5 Shift operators

Syntax:

```
Syntax

shift-operator: one of
    dash shl
    dash shr

dash:
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)
```

Description:

The shift left (`-shl`) operator and shift right (`-shr`) operator convert the value designed by the left operand to an integer type and the value designated by the right operand to `int`, if necessary, using the usual arithmetic conversions ([§6.15](#)).

The shift left operator shifts the left operand left by a number of bits computed as described below. The low-order empty bit positions are set to zero.

The shift right operator shifts the left operand right by a number of bits computed as described below. The low-order bits of the left operand are discarded, the remaining bits shifted right. When the left operand is a signed value, the high-order empty bit positions are set to zero if the left operand is non-negative and set to one if the left operand is negative. When the left operand is an unsigned value, the high-order empty bit positions are set to zero.

When the left operand has type `int`, the shift count is given by the low-order five bits of the right operand. When the right operand has type `long`, the shift count is given by the low-order six bits of the right operand.

Examples:

```
PowerShell
```

```
0x0408 -shl 1          # int with value 0x0810
0x0408 -shr 3          # int with value 0x0081
0x100000000 -shr 0xffff81 # long with value 0x80000000
```

7.9 Bitwise operators

Syntax:

Syntax

```
bitwise-expression:
    unary-expression -band new-lines~opt~ unary-expression
    unary-expression -bor new-lines~opt~ unary-expression
    unary-expression -bxor new-lines~opt~ unary-expression
```

Description:

The bitwise AND operator `-band`, the bitwise OR operator `-bor`, and the bitwise XOR operator `-bxor` convert the values designated by their operands to integer types, if necessary, using the usual arithmetic conversions ([§6.15](#)). After conversion, if both values have type int that is the type of the result. Otherwise, if both values have type long, that is the type of the result. If one value has type int and the other has type long, the type of the result is long. Otherwise, the expression is ill formed. The result is the bitwise AND, bitwise OR, or bitwise XOR, respectively, of the possibly converted operand values.

These operators are left associative. They are commutative if neither operand contains a side effect.

Examples:

PowerShell

```
0x0F0F -band 0xFE      # int with value 0xE
0x0F0F -band 0xFEL     # long with value 0xE
0x0F0F -band 14.6       # long with value 0xF

0x0F0F -bor 0xFE       # int with value 0xFFFF
0x0F0F -bor 0xFEL      # long with value 0xFFFF
0x0F0F -bor 14.40D      # long with value 0xF0F

0x0F0F -bxor 0xFE      # int with value 0xFF1
0x0F0F -bxor 0xFEL     # long with value 0xFF1
0x0F0F -bxor 14.40D     # long with value 0xF01
0x0F0F -bxor 14.6       # long with value 0xF00
```

7.10 Logical operators

Syntax:

```
Syntax

logical-expression:
    unary-expression -and new-lines~opt~ unary-expression
    unary-expression -or new-lines~opt~ unary-expression
    unary-expression -xor new-lines~opt~ unary-expression
```

Description:

The logical AND operator `-and` converts the values designated by its operands to `bool`, if necessary ([§6.2](#)). The result is the logical AND of the possibly converted operand values, and has type `bool`. If the left operand evaluates to False the right operand is not evaluated.

The logical OR operator `-or` converts the values designated by its operands to `bool`, if necessary ([§6.2](#)). The result is the logical OR of the possibly converted operand values, and has type `bool`. If the left operand evaluates to True the right operand is not evaluated.

The logical XOR operator `-xor` converts the values designated by its operands to `bool` ([§6.2](#)). The result is the logical XOR of the possibly converted operand values, and has type `bool`.

These operators are left associative.

Examples:

```
PowerShell

$j = 10
$k = 20
($j -gt 5) -and (++$k -lt 15) # True -and False -> False
($j -gt 5) -and ($k -le 21)    # True -and True -> True
($j++ -gt 5) -and ($j -le 10)   # True -and False -> False
($j -eq 5) -and (++$k -gt 15)   # False -and True -> False

$j = 10
$k = 20
($j++ -gt 5) -or (++$k -lt 15) # True -or False -> True
($j -eq 10) -or ($k -gt 15)    # False -or True -> True
($j -eq 10) -or (++$k -le 20)   # False -or False -> False

$j = 10
```

```
$k = 20
($j++ -gt 5) -xor (++$k -lt 15) # True -xor False -> True
($j -eq 10) -xor ($k -gt 15)     # False -xor True -> True
($j -gt 10) -xor (++$k -le 25)   # True -xor True -> False
```

7.11 Assignment operators

Syntax:

```
Syntax

assignment-expression:
    expression assignment-operator statement

assignment-operator: *one of
    =    dash =    +=    *=    /=    %=
```

Description:

An assignment operator stores a value in the writable location designated by *expression*. For a discussion of *assignment-operator* `=` see §7.11.1. For a discussion of all other assignment operators see §7.11.2.

An assignment expression has the value designated by *expression* after the assignment has taken place; however, that assignment expression does not itself designate a writable location. If *expression* is type-constrained (§5.3), the type used in that constraint is the type of the result; otherwise, the type of the result is the type after the usual arithmetic conversions (§6.15) have been applied.

This operator is right associative.

7.11.1 Simple assignment

Description:

In *simple assignment* (`=`), the value designated by *statement* replaces the value stored in the writable location designated by *expression*. However, if *expression* designates a non-existent key in a Hashtable, that key is added to the Hashtable with an associated value of the value designated by *statement*.

As shown by the grammar, *expression* may designate a comma-separated list of writable locations. This is known as *multiple assignment*. *statement* designates a list of one or more comma-separated values. The commas in either operand list are part of the multiple-assignment syntax and do *not* represent the binary comma operator. Values are

taken from the list designated by *statement*, in lexical order, and stored in the corresponding writable location designated by *expression*. If the list designated by *statement* has fewer values than there are *expression* writable locations, the excess locations take on the value `$null`. If the list designated by *statement* has more values than there are *expression* writable locations, all but the right-most *expression* location take on the corresponding *statement* value and the right-most *expression* location becomes an unconstrained 1-dimensional array with all the remaining *statement* values as elements.

For statements that have values ([§8.1.2](#)), *statement* can be a statement.

Examples:

PowerShell

```
$a = 20; $b = $a + 12L          # $b has type long, value 22
$hypot = [Math]::Sqrt(3*3 + 4*4)    # type double, value 5
$a = $b = $c = 10.20D           # all have type decimal, value 10.20
$a = (10,20,30),(1,2)           # type [Object[]], Length 2
[int]$x = 10.6                  # type int, value 11
[long]$x = "0xabc"              # type long, value 0xabc
$a = [float]                     # value type literal [float]
$i,$j,$k = 10,"red",$true       # $i is 10, $j is "red", $k is True
$i,$j = 10,"red",$true          # $i is 10, $j is [Object[]], Length 2
$i,$j = (10,"red"),$true        # $i is [Object[]], Length 2, $j is True
$i,$j,$k = 10                   # $i is 10, $j is $null, $k is $null

$h = @{}
[int] $h.Lower, [int] $h.Upper = -split "10 100"

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }
$h1.Dept = "Finance"            # adds element Finance
$h1["City"] = "New York"         # adds element City

[int]$Variable:v = 123.456        # v takes on the value 123
${E:output.txt} = "a"             # write text to the given file
$Env:MyPath = "X:\data\file.txt"  # define the environment variable
$Function:F = { param ($a, $b) "Hello there, $a, $b" }
F 10 "red"                      # define and invoke a function
function Demo { "Hi there from inside Demo" }
$Alias:A = "Demo"                # create alias for function Demo
A                                # invoke function Demo via the alias
```

7.11.2 Compound assignment

Description:

A *compound assignment* has the form `E1 op= E2`, and is equivalent to the simple assignment expression `E1 = E1 op (E2)` except that in the compound assignment case the expression *E1* is evaluated only once. If *expression* is type-constrained (§5.3), the type used in that constraint is the type of the result; otherwise, the type of the result is determined by *op*. For `*=`, see §7.6.1, §7.6.2, §7.6.3; for `/=`, see §7.6.4; for `%=`, see §7.6.5; for `+=`, see §7.7.1, §7.7.2, §7.7.3; for `-=`, see §7.7.5.

ⓘ Note

An operand designating an unconstrained value of numeric type may have its type changed by an assignment operator when the result is stored.

Examples:

PowerShell

```
$a = 1234; $a *= (3 + 2) # type is int, value is 1234 * (3 + 2)
$b = 10,20,30           # $b[1] has type int, value 20
$b[1] /= 6              # $b[1] has type double, value 3.33...

$i = 0
$b = 10,20,30           # side effect evaluated only once
$b[++$i] += 2

[int]$Variable:v = 10    # v takes on the value 10
$Variable:v -= 3         # 3 is subtracted from v

${E:output.txt} = "a"    # write text to the given file
${E:output.txt} += "b"    # append text to the file giving ab
${E:output.txt} *= 4      # replicate ab 4 times giving ababab
```

7.12 Redirection operators

Syntax:

Syntax

```
pipeline:
  expression redirections~opt~ pipeline-tail~opt~
  command verbatim-command-argument~opt~ pipeline-tail~opt~

redirections:
  redirection
  redirections redirection

redirection:
```

```

merging-redirection-operator
file-redirection-operator redirected-file-name

redirected-file-name:
  command-argument
  primary-expression

file-redirection-operator: one of
  >    >>   2>   2>>   3>   3>>   4>   4>>
  5>   5>>   6>   6>>   >    >>    <

merging-redirection-operator: one of
  >&1   2>&1   3>&1   4>&1   5>&1   6>&1
  >&2   1>&2   3>&2   4>&2   5>&2   6>&2

```

Description:

The redirection operator `>` takes the standard output from the pipeline and redirects it to the location designated by *redirected-file-name*, overwriting that location's current contents.

The redirection operator `>>` takes the standard output from the pipeline and redirects it to the location designated by *redirected-file-name*, appending to that location's current contents, if any. If that location does not exist, it is created.

The redirection operator with the form `n>` takes the output of stream *n* from the pipeline and redirects it to the location designated by *redirected-file-name*, overwriting that location's current contents.

The redirection operator with the form `n>>` takes the output of stream *n* from the pipeline and redirects it to the location designated by *redirected-file-name*, appending to that location's current contents, if any. If that location does not exist, it is created.

The redirection operator with the form `m>&n` writes output from stream *m* to the same location as stream *n*.

The following are the valid streams:

[] Expand table

Stream	Description
1	Standard output stream
2	Error output stream
3	Warning output stream

Stream	Description
4	Verbose output stream
5	Debug output stream
*	Standard output, error output, warning output, verbose output, and debug output streams

The redirection operators `1>&2`, `6>`, `6>>` and `<` are reserved for future use.

If on output the value of *redirected-file-name* is `$null`, the output is discarded.

Ordinarily, the value of an expression containing a top-level side effect is not written to the pipeline unless that expression is enclosed in a pair of parentheses. However, if such an expression is the left operand of an operator that redirects standard output, the value is written.

Examples:

PowerShell

```
$i = 200                      # pipeline gets nothing
$i                         # pipeline gets result
$i > output1.txt           # result redirected to named file
++$i >> output1.txt        # result appended to named file
type file1.txt 2> error1.txt # error output redirected to named file
type file2.txt 2>> error1.txt # error output appended to named file
dir -Verbose 4> verbose1.txt # verbose output redirected to named file

# Send all output to output2.txt
dir -Verbose -Debug -WarningAction Continue *> output2.txt

# error output redirected to named file, verbose output redirected
# to the same location as error output
dir -Verbose 4>&2 2> error2.txt
```

8. Statements

Article • 07/15/2024

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

8.1 Statement blocks and lists

Syntax:

Tip

The `~opt~` notation in the syntax definitions indicates that the lexical entity is optional in the syntax.

Syntax

```
statement-block:  
    new-lines~opt~ { statement-list~opt~ new-lines~opt~ }  
  
statement-list:  
    statement  
    statement-list statement  
  
statement:  
    if-statement
```

```
label~opt~ labeled-statement
function-statement
flow-control-statement statement-terminator
trap-statement
try-statement
data-statement
inlinescript-statement
parallel-statement
sequence-statement
pipeline statement-terminator

statement-terminator:
;
new-line-character
```

Description:

A *statement* specifies some sort of action that is to be performed. Unless indicated otherwise within this clause, statements are executed in lexical order.

A *statement-block* allows a set of statements to be grouped into a single syntactic unit.

8.1.1 Labeled statements

Syntax:

Syntax
<pre>labeled-statement: switch-statement foreach-statement for-statement while-statement do-statement</pre>

Description:

An iteration statement ([§8.4](#)) or a switch statement ([§8.6](#)) may optionally be preceded immediately by one statement label, *label*. A statement label is used as the optional target of a break ([§8.5.1](#)) or continue ([§8.5.2](#)) statement. However, a label does not alter the flow of control.

White space is not permitted between the colon (:) and the token that follows it.

Examples:

PowerShell

```

:go_here while ($j -le 100) {
    # ...
}

:labelA
for ($i = 1; $i -le 5; ++$i) {
    :labelB
    for ($j = 1; $j -le 3; ++$j) {
        :labelC
        for ($k = 1; $k -le 2; ++$k) {
            # ...
        }
    }
}

```

8.1.2 Statement values

The value of a statement is the cumulative set of values that it writes to the pipeline. If the statement writes a single scalar value, that is the value of the statement. If the statement writes multiple values, the value of the statement is that set of values stored in elements of an unconstrained 1-dimensional array, in the order in which they were written. Consider the following example:

```
$v = for ($i = 10; $i -le 5; ++$i) { }
```

There are no iterations of the loop and nothing is written to the pipeline. The value of the statement is `$null`.

```
$v = for ($i = 1; $i -le 5; ++$i) { }
```

Although the loop iterates five times nothing is written to the pipeline. The value of the statement is `$null`.

```
$v = for ($i = 1; $i -le 5; ++$i) { $i }
```

The loop iterates five times each time writing to the pipeline the `int` value `$i`. The value of the statement is `object[]` of Length 5.

```
$v = for ($i = 1; $i -le 5; ) { ++$i }
```

Although the loop iterates five times nothing is written to the pipeline. The value of the statement is `$null`.

```
$v = for ($i = 1; $i -le 5; ) { (++$i) }
```

The loop iterates five times with each value being written to the pipeline. The value of the statement is `object[]` of Length 5.

```
$i = 1; $v = while ($i++ -lt 2) { $i }
```

The loop iterates once. The value of the statement is the `int` with value 2.

Here are some other examples:

PowerShell

```
# if $count is not currently defined then define it with int value 10
$count = if ($count -eq $null) { 10 } else { $count }

$i = 1
$v = while ($i -le 5) {
    $i                         # $i is written to the pipeline
    if ($i -band 1) {

        "odd"                  # conditionally written to the pipeline

    }
    ++$i                      # not written to the pipeline
}

# $v is object[], Length 8, value 1,"odd",2,3,"odd",4,5,"odd"
```

8.2 Pipeline statements

Syntax:

Syntax

```
pipeline:
    assignment-expression
    expression redirections~opt~ pipeline-tail~opt~
    command verbatim-command-argument~opt~ pipeline-tail~opt~

assignment-expression:
    expression assignment-operator statement

pipeline-tail:
    | new-lines~opt~ command
    | new-lines~opt~ command pipeline-tail

command:
    command-name command-elements~opt~
    command-invocation-operator command-module~opt~ command-name-expr
    command-elements~opt~
```

```

command-invocation-operator: one of
& .

command-module:
    primary-expression

command-name:
    generic-token
    generic-token-with-subexpr

generic-token-with-subexpr:
    No whitespace is allowed between ) and command-name.
    generic-token-with-subexpr-start statement-list~opt~ )

command-namecommand-name-expr:
    command-name

primary-expressioncommand-elements:
    command-element
    command-elements command-element

command-element:
    command-parameter
    command-argument
    redirection

command-argument:
    command-name-expr

verbatim-command-argument:
    --% verbatim-command-argument-chars

```

Description:

redirections is discussed in [§7.12](#); *assignment-expression* is discussed in [§7.11](#); and the *command-invocation-operator* dot (.) is discussed in [§3.5.5](#). For a discussion of argument-to-parameter mapping in command invocations, see [§8.14](#).

The first command in a *pipeline* is an expression or a command invocation. Typically, a command invocation begins with a *command-name*, which is usually a bare identifier. *command-elements* represents the argument list to the command. A newline or n unescaped semicolon terminates a pipeline.

A command invocation consists of the command's name followed by zero or more arguments. The rules governing arguments are as follows:

- An argument that is not an expression, but which contains arbitrary text without unescaped white space, is treated as though it were double quoted. Letter case is preserved.

- Variable substitution and sub-expression expansion ([§2.3.5.2](#)) takes place inside *expandable-string-literals* and *expandable-here-string-literals*.
- Text inside quotes allows leading, trailing, and embedded white space to be included in the argument's value. [Note: The presence of whitespace in a quoted argument does not turn a single argument into multiple arguments. *end note*]
- Putting parentheses around an argument causes that expression to be evaluated with the result being passed instead of the text of the original expression.
- To pass an argument that looks like a switch parameter ([§2.3.4](#)) but is not intended as such, enclose that argument in quotes.
- When specifying an argument that matches a parameter having the `[switch]` type constraint ([§8.10.5](#)), the presence of the argument name on its own causes that parameter to be set to `$true`. However, the parameter's value can be set explicitly by appending a suffix to the argument. For example, given a type constrained parameter *p*, an argument of `-p:$true` sets *p* to True, while `-p:$false` sets *p* to False.
- An argument of `--` indicates that all arguments following it are to be passed in their actual form as though double quotes were placed around them.
- An argument of `--%` indicates that all arguments following it are to be passed with minimal parsing and processing. This argument is called the verbatim parameter. Arguments after the verbatim parameter are not PowerShell expressions even if they are syntactically valid PowerShell expressions.

If the command type is Application, the parameter `--%` is not passed to the command. The arguments after `--%` have any environment variables (strings surrounded by `%`) expanded. For example:

PowerShell

```
echoargs.exe --% "%path%" # %path% is replaced with the value $Env:path
```

The order of evaluation of arguments is unspecified.

For information about parameter binding see [§8.14](#). For information about name lookup see [§3.8](#).

Once argument processing has been completed, the command is invoked. If the invoked command terminates normally ([§8.5.4](#)), control reverts to the point in the script or function immediately following the command invocation. For a description of the

behavior on abnormal termination see `break` ([§8.5.1](#)), `continue` ([§8.5.2](#)), `throw` ([§8.5.3](#)), `exit` ([§8.5.5](#)), `try` ([§8.7](#)), and `trap` ([§8.8](#)).

Ordinarily, a command is invoked by using its name followed by any arguments. However, the command-invocation operator, `&`, can be used. If the command name contains unescaped white space, it must be quoted and invoked with this operator. As a script block has no name, it too must be invoked with this operator. For example, the following invocations of a command call `Get-Factorial` are equivalent:

PowerShell

```
Get-Factorial 5
& Get-Factorial 5
& "Get-Factorial" 5
```

Direct and indirect recursive function calls are permitted. For example,

PowerShell

```
function Get-Power([int]$x, [int]$y) {
    if ($y -gt 0) { return $x * (Get-Power $x (--$y)) }
    else { return 1 }
}
```

Examples:

PowerShell

```
New-Object 'int[,]' 3,2
New-Object -ArgumentList 3,2 -TypeName 'int[,]'

dir E:\PowerShell\Scripts\*statement*.ps1 | ForEach-Object {$_.Length}

dir E:\PowerShell\Scripts\*.ps1 |
    Select-String -List "catch" |
        Format-Table Path, LineNumber -AutoSize
```

8.3 The if statement

Syntax:

Syntax

```
if-statement:
    if new-lines~opt~ ( new-lines~opt~ pipeline new-lines~opt~ ) statement-
```

```

block
    elseif-clauses~opt~ else-clause~opt~

elseif-clauses:
    elseif-clause
    elseif-clauses elseif-clause

elseif-clause:
    new-lines~opt~ elseif new-lines~opt~ ( new-lines~opt~ pipeline new-
lines~opt~ ) statement-block

else-clause:
    new-lines~opt~ else statement-block

```

Description:

The *pipeline* controlling expressions must have type bool or be implicitly convertible to that type. The *else-clause* is optional. There may be zero or more *elseif-clauses*.

If the top-level *pipeline* tests True, then its *statement-block* is executed and execution of the statement terminates. Otherwise, if an *elseif-clause* is present, if its *pipeline* tests True, then its *statement-block* is executed and execution of the statement terminates. Otherwise, if an *else-clause* is present, its *statement-block* is executed.

Examples:

PowerShell

```

$grade = 92
if ($grade -ge 90) { "Grade A" }
elseif ($grade -ge 80) { "Grade B" }
elseif ($grade -ge 70) { "Grade C" }
elseif ($grade -ge 60) { "Grade D" }
else { "Grade F" }

```

8.4 Iteration statements

8.4.1 The while statement

Syntax:

Syntax

```

while-statement:
    while new-lines~opt~ ( new-lines~opt~ while-condition new-lines~opt~ )
    statement-block

```

```
while-condition:  
    new-lines~opt~ pipeline
```

Description:

The controlling expression *while-condition* must have type bool or be implicitly convertible to that type. The loop body, which consists of *statement-block*, is executed repeatedly until the controlling expression tests False. The controlling expression is evaluated before each execution of the loop body.

Examples:

PowerShell

```
$i = 1  
while ($i -le 5) {  
    "{0,1}`t{1,2}" -f $i, ($i*$i)  
    ++$i  
}
```

8.4.2 The do statement

Syntax:

Syntax

```
do-statement:  
    do statement-block new-lines~opt~ while new-lines~opt~ ( while-condition  
new-lines~opt~ )  
        do statement-block new-lines~opt~ until new-lines~opt~ ( while-condition  
new-lines~opt~ )  
  
while-condition:  
    new-lines~opt~ pipeline
```

Description:

The controlling expression *while-condition* must have type bool or be implicitly convertible to that type. In the while form, the loop body, which consists of *statement-block*, is executed repeatedly while the controlling expression tests True. In the until form, the loop body is executed repeatedly until the controlling expression tests True. The controlling expression is evaluated after each execution of the loop body.

Examples:

PowerShell

```

$i = 1
do {
    "{0,1}`t{1,2}" -f $i, ($i * $i)
}
while (++$i -le 5)                      # loop 5 times

$i = 1
do {
    "{0,1}`t{1,2}" -f $i, ($i * $i)
}
until (++$i -gt 5)                      # loop 5 times

```

8.4.3 The for statement

Syntax:

Syntax

```

for-statement:
    for new-lines~opt~ (
        new-lines~opt~ for-initializer~opt~ statement-terminator
        new-lines~opt~ for-condition~opt~ statement-terminator
        new-lines~opt~ for-iterator~opt~
        new-lines~opt~ ) statement-block

    for new-lines~opt~ (
        new-lines~opt~ for-initializer~opt~ statement-terminator
        new-lines~opt~ for-condition~opt~
        new-lines~opt~ ) statement-block

    for new-lines~opt~ (
        new-lines~opt~ for-initializer~opt~
        new-lines~opt~ ) statement-block

for-initializer:
    pipeline

for-condition:
    pipeline

for-iterator:
    pipeline

```

Description:

The controlling expression *for-condition* must have type bool or be implicitly convertible to that type. The loop body, which consists of *statement-block*, is executed repeatedly while the controlling expression tests True. The controlling expression is evaluated before each execution of the loop body.

Expression *for-initializer* is evaluated before the first evaluation of the controlling expression. Expression *for-initializer* is evaluated for its side effects only; any value it produces is discarded and is not written to the pipeline.

Expression *for-iterator* is evaluated after each execution of the loop body. Expression *for-iterator* is evaluated for its side effects only; any value it produces is discarded and is not written to the pipeline.

If expression *for-condition* is omitted, the controlling expression tests True.

Examples:

PowerShell

```
for ($i = 5; $i -ge 1; --$i) { # loop 5 times
    "{0,1}`t{1,2}" -f $i, ($i * $i)
}

$i = 5
for (; $i -ge 1; ) { # equivalent behavior
    "{0,1}`t{1,2}" -f $i, ($i * $i)
    --$i
}
```

8.4.4 The foreach statement

Syntax:

Syntax

```
foreach-statement:
    foreach new-lines~opt~ foreach-parameter~opt~ new-lines~opt~
        ( new-lines~opt~ variable new-lines~opt~ *in* new-lines~opt~
pipeline
    new-lines~opt~ ) statement-block

foreach-parameter:
    -parallel
```

Description:

The loop body, which consists of *statement-block*, is executed for each element designated by the variable *variable* in the collection designated by *pipeline*. The scope of *variable* is not limited to the foreach statement. As such, it retains its final value after the loop body has finished executing. If *pipeline* designates a scalar (excluding the value

`$null`) instead of a collection, that scalar is treated as a collection of one element. If `pipeline` designates the value `$null`, `pipeline` is treated as a collection of zero elements.

If the *foreach-parameter* `-parallel` is specified, the behavior is implementation defined.

The *foreach-parameter* `-parallel` is only allowed in a workflow ([§8.10.2](#)).

Every foreach statement has its own enumerator, `$foreach` ([§2.3.2.2](#), [§4.5.16](#)), which exists only while that loop is executing.

The objects produced by `pipeline` are collected before *statement-block* begins to execute. However, with the [ForEach-Object cmdlet](#), *statement-block* is executed on each object as it is produced.

Examples:

PowerShell

```
$a = 10, 53, 16, -43
foreach ($e in $a) {
    ...
}
$e # the int value -43

foreach ($e in -5..5) {
    ...
}

foreach ($t in [byte], [int], [long]) {
    $t::MaxValue # get static property
}

foreach ($f in Get-ChildItem *.txt) {
    ...
}

$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }
foreach ($e in $h1.Keys) {
    "Key is " + $e + ", Value is " + $h1[$e]
}
```

8.5 Flow control statements

Syntax:

Syntax

```
flow-control-statement:
    break label-expression~opt~
    continue label-expression~opt~
    throw pipeline~opt~
    return pipeline~opt~
    exit pipeline~opt~

label-expression:
    simple-name
    unary-expression
```

Description:

A flow-control statement causes an unconditional transfer of control to some other location.

8.5.1 The `break` statement

Description:

A `break` statement with a *label-expression* is referred to as a *labeled break statement*. A `break` statement without a *label-expression* is referred to as an *unlabeled break statement*.

Outside a trap statement, an unlabeled `break` statement directly within an iteration statement ([§8.4](#)) terminates execution of that smallest enclosing iteration statement. An unlabeled `break` statement directly within a switch statement ([§8.6](#)) terminates pattern matching for the current switch's *switch-condition*. See ([§8.8](#)) for details of using `break` from within a trap statement.

An iteration statement or a switch statement may optionally be preceded immediately by one statement label ([§8.1.1](#)). Such a statement label may be used as the target of a labeled `break` statement, in which case, that statement terminates execution of the targeted enclosing iteration statement.

A labeled `break` need not be resolved in any local scope; the search for a matching label may continue up the calling stack even across script and function-call boundaries. If no matching label is found, the current command invocation is terminated.

The name of the label designated by *label-expression* need not have a constant value.

If *label-expression* is a *unary-expression*, it is converted to a string.

Examples:

PowerShell

```
$i = 1
while ($true) { # infinite loop
    if ($i * $i -gt 100) {
        break # break out of current while loop
    }
    ++$i
}

$lab = "go_here"
:go_here
for ($i = 1; ; ++$i) {
    if ($i * $i -gt 50) {
        break $lab # use a string value as target
    }
}

:labelA
for ($i = 1; $i -le 2; $i++) {

    :labelB
    for ($j = 1; $j -le 2; $j++) {

        :labelC
        for ($k = 1; $k -le 3; $k++) {
            if (...) { break labelA }
        }
    }
}
```

8.5.2 The continue statement

Description:

A `continue` statement with a *label-expression* is referred to as a *labeled continue statement*. A `continue` statement without a *label-expression* is referred to as an *unlabeled continue statement*.

The use of `continue` from within a trap statement is discussed in [§8.8](#).

An unlabeled `continue` statement within a loop terminates execution of the current loop and transfers control to the closing brace of the smallest enclosing iteration statement ([§8.4](#)). An unlabeled `continue` statement within a switch terminates execution of the current `switch` iteration and transfers control to the smallest enclosing `switch`'s *switch-condition* ([§8.6](#)).

An iteration statement or a `switch` statement ([§8.6](#)) may optionally be preceded immediately by one statement label ([§8.1.1](#)). Such a statement label may be used as the target of an enclosed labeled `continue` statement, in which case, that statement terminates execution of the current loop or `switch` iteration, and transfers control to the targeted enclosing iteration or `switch` statement label.

A labeled `continue` need not be resolved in any local scope; the search for a matching label may `continue` up the calling stack even across script and function-call boundaries. If no matching label is found, the current command invocation is terminated.

The name of the label designated by *label-expression* need not have a constant value.

If *label-expression* is a *unary-expression*, it is converted to a string.

Examples:

PowerShell

```
$i = 1
while (...) {
    ...
    if (...) {
        continue # start next iteration of current loop
    }
    ...
}

$lab = "go_here"
:go_here
for (...; ...; ...) {
    if (...) {
        continue $lab # start next iteration of labeled loop
    }
}

:labelA
for ($i = 1; $i -le 2; $i++) {

    :labelB
    for ($j = 1; $j -le 2; $j++) {

        :labelC
        for ($k = 1; $k -le 3; $k++) {
            if (...) { continue labelB }
        }
    }
}
```

8.5.3 The throw statement

Description:

An exception is a way of handling a system- or application-level error condition. The `throw` statement raises an exception. (See [§8.7](#) for a discussion of exception handling.)

If *pipeline* is omitted and the `throw` statement is not in a *catch-clause*, the behavior is implementation defined. If *pipeline* is present and the `throw` statement is in a *catch-clause*, the exception that was caught by that *catch-clause* is re-thrown after any *finally-clause* associated with the *catch-clause* is executed.

If *pipeline* is present, the type of the exception thrown is implementation defined.

When an exception is thrown, control is transferred to the first catch clause in an enclosing try statement that can handle the exception. The location at which the exception is thrown initially is called the *throw point*. Once an exception is thrown the steps described in [§8.7](#) are followed repeatedly until a catch clause that matches the exception is found or none can be found.

Examples:

PowerShell

```
throw
throw 100
throw "No such record in file"
```

If *pipeline* is omitted and the `throw` statement is not from within a *catch-clause*, the text "ScriptHalted" is written to the pipeline, and the type of the exception raised is `System.Management.Automation.RuntimeException`.

If *pipeline* is present, the exception raised is wrapped in an object of type `System.Management.Automation.RuntimeException`, which includes information about the exception as a `System.Management.Automation.ErrorRecord` object (accessible via `$_.`).

Example 1: `throw 123` results in an exception of type `RuntimeException`. From within the catch block, `$_.TargetObject` contains the object wrapped inside, in this case, a `System.Int32` with value 123.

Example 2: `throw "xxx"` results in an exception of type `RuntimeException`. From within the catch block, `$_.TargetObject` contains the object wrapped inside, in this case, a `System.String` with value "xxx".

Example 3: `throw 10,20` results in an exception of type **RuntimeException**. From within the catch block, `$_._TargetObject` contains the object wrapped inside, in this case, a `System.Object[]`, an unconstrained array of two elements with the `System.Int32`` values 10 and 20.

8.5.4 The return statement

Description:

The `return` statement writes to the pipeline the value(s) designated by *pipeline*, if any, and returns control to the function or script's caller. A function or script may have zero or more `return` statements.

If execution reaches the closing brace of a function an implied `return` without *pipeline* is assumed.

The `return` statement is a bit of "syntactic sugar" to allow programmers to express themselves as they can in other languages; however, the value returned from a function or script is actually all of the values written to the pipeline by that function or script plus any value(s) specified by *pipeline*. If only a scalar value is written to the pipeline, its type is the type of the value returned; otherwise, the return type is an unconstrained 1-dimensional array containing all the values written to the pipeline.

Examples:

PowerShell

```
function Get-Factorial ($v) {
    if ($v -eq 1) {
        return 1 # return is not optional
    }

    return $v * (Get-Factorial ($v - 1)) # return is optional
}
```

The caller to `Get-Factorial` gets back an `int`.

PowerShell

```
function Test {
    "text1" # "text1" is written to the pipeline
    #
    "text2" # "text2" is written to the pipeline
    #
}
```

```
    return 123 # 123 is written to the pipeline
}
```

The caller to `Test` gets back an unconstrained 1-dimensional array of three elements.

8.5.5 The exit statement

Description:

The `exit` statement terminates the current script and returns control and an *exit code* to the host environment or the calling script. If *pipeline* is provided, the value it designates is converted to `int`, if necessary. If no such conversion exists, or if *pipeline* is omitted, the `int` value zero is returned.

Examples:

PowerShell

```
exit $count # terminate the script with some accumulated count
```

8.6 The switch statement

Syntax:

Syntax

```
switch-statement:
    switch new-lines~opt~ switch-parameters~opt~ switch-condition switch-
body

switch-parameters:
    switch-parameter
    switch-parameters switch-parameter

switch-parameter:
    -Regex
    -Wildcard
    -Exact
    -CaseSensitive
    -Parallel

switch-condition:
    ( new-lines~opt~ pipeline new-lines~opt~ )
    -File new-lines~opt~ switch-filename

switch-filename:
    command-argument
```

```
primary-expression

switch-body:
    new-lines~opt~ { new-lines~opt~ switch-clauses }

switch-clauses:
    switch-clause
    switch-clauses switch-clause

switch-clause:
    switch-clause-condition statement-block statement-terminators~opt~

switch-clause-condition:
    command-argument
    primary-expression
```

Description:

If *switch-condition* designates a single value, control is passed to one or more matching pattern statement blocks. If no patterns match, some default action can be taken.

A switch must contain one or more *switch-clauses*, each starting with a pattern (a *non-default switch clause*), or the keyword `default` (a *default switch clause*). A switch must contain zero or one `default` switch clauses, and zero or more non-default switch clauses. Switch clauses may be written in any order.

Multiple patterns may have the same value. A pattern need not be a literal, and a switch may have patterns with different types.

If the value of *switch-condition* matches a pattern value, that pattern's *statement-block* is executed. If multiple pattern values match the value of *switch-condition*, each matching pattern's *statement-block* is executed, in lexical order, unless any of those *statement-blocks* contains a `break` statement ([§8.5.1](#)).

If the value of *switch-condition* does not match any pattern value, if a `default` switch clause exists, its *statement-block* is executed; otherwise, pattern matching for that *switch-condition* is terminated.

Switches may be nested, with each switch having its own set of switch clauses. In such instances, a switch clause belongs to the innermost switch currently in scope.

On entry to each *statement-block*, `$_` is automatically assigned the value of the *switch-condition* that caused control to go to that *statement-block*. `$_` is also available in that *statement-block's switch-clause-condition*.

Matching of non-strings is done by testing for equality ([§7.8.1](#)).

If the matching involves strings, by default, the comparison is case-insensitive. The presence of the *switch-parameter* `-CaseSensitive` makes the comparison case-sensitive.

A pattern may contain wildcard characters ([§3.15](#)), in which case, wildcard string comparisons are performed, but only if the *switch-parameter* `-Wildcard` is present. By default, the comparison is case-insensitive.

A pattern may contain a regular expression ([§3.16](#)), in which case, regular expression string comparisons are performed, but only if the *switch-parameter* `-Regex` is present. By default, the comparison is case-insensitive. If `-Regex` is present and a pattern is matched, `$Matches` is defined in the *switch-clause statement-block* for that pattern.

A *switch-parameter* may be abbreviated; any distinct leading part of a parameter may be used. For example, `-Regex`, `-Rege`, `-Reg`, `-Re`, and `-R` are equivalent.

If conflicting *switch-parameters* are specified, the lexically final one prevails. The presence of `-Exact` disables `-Regex` and `-Wildcard`; it has no affect on `-Case`, however.

If the *switch-parameter* `-Parallel` is specified, the behavior is implementation defined.

The *switch-parameter* `-Parallel` is only allowed in a workflow ([§8.10.2](#)).

If a pattern is a *script-block-expression*, that block is evaluated and the result is converted to bool, if necessary. If the result has the value `$true`, the corresponding *statement-block* is executed; otherwise, it is not.

If *switch-condition* designates multiple values, the switch is applied to each value in lexical order using the rules described above for a *switch-condition* that designates a single value. Every switch statement has its own enumerator, `$switch` ([§2.3.2.2](#), [§4.5.16](#)), which exists only while that switch is executing.

A switch statement may have a label, and it may contain labeled and unlabeled break ([§8.5.1](#)) and continue ([§8.5.2](#)) statements.

If *switch-condition* is `-File` *switch-filename*, instead of iterating over the values in an expression, the switch iterates over the values in the file designated by *switch-filename*. The file is read a line at a time with each line comprising a value. Line terminator characters are not included in the values.

Examples:

PowerShell

```
$s = "ABC def`nghi`tjkl`fmno @##"  
$charCount = 0; $pageCount = 0; $lineCount = 0; $otherCount = 0
```

```

for ($i = 0; $i -lt $s.Length; ++$i) {
    ++$charCount
    switch ($s[$i]) {
        "`n" { ++$lineCount }
        "`f" { ++$pageCount }
        "`t" { }
        " " { }
        default { ++$otherCount }
    }
}

switch -Wildcard ("abc") {
    a* { "a*, $_" }
    ?B? { "?B?, $_" }
    default { "default, $_" }
}

switch -Regex -CaseSensitive ("abc") {
    ^a* { "a*" }
    ^A* { "A*" }
}

switch (0, 1, 19, 20, 21) {
    { $_ -lt 20 } { "-lt 20" }
    { $_ -band 1 } { "Odd" }
    { $_ -eq 19 } { "-eq 19" }
    default { "default" }
}

```

8.7 The try/finally statement

Syntax:

Syntax

```

try-statement:
    try statement-block catch-clauses
    try statement-block finally-clause
    try statement-block catch-clauses finally-clause

catch-clauses:
    catch-clause
    catch-clauses catch-clause

catch-clause:
    new-lines~opt~ catch catch-type-list~opt~
    statement-block

catch-type-list:
    new-lines~opt~ type-literal
    catch-type-list new-lines~opt~, new-lines~opt~

```

```
type-literalfinally-clause:  
    new-lines~opt~ finally statement-block
```

Description:

The `try` statement provides a mechanism for catching exceptions that occur during execution of a block. The `try` statement also provides the ability to specify a block of code that is always executed when control leaves the `try` statement. The process of raising an exception via the `throw` statement is described in §8.5.3.

A *try block* is the *statement-block* associated with the `try` statement. A *catch block* is the *statement-block* associated with a *catch-clause*. A *finally block* is the *statement-block* associated with a *finally-clause*.

A *catch-clause* without a *catch-type-list* is called a *general catch clause*.

Each *catch-clause* is an *exception handler*, and a *catch-clause* whose *catch-type-list* contains the type of the raised exception is a *matching catch clause*. A general catch clause matches all exception types.

Although *catch-clauses* and *finally-clause* are optional, at least one of them must be present.

The processing of a thrown exception consists of evaluating the following steps repeatedly until a catch clause that matches the exception is found.

- In the current scope, each `try` statement that encloses the throw point is examined. For each `try` statement *S*, starting with the innermost `try` statement and ending with the outermost `try` statement, the following steps are evaluated:
 - If the `try` block of *S* encloses the throw point and if *S* has one or more catch clauses, the catch clauses are examined in lexical order to locate a suitable handler for the exception. The first catch clause that specifies the exception type or a base type of the exception type is considered a match. A general catch clause is considered a match for any exception type. If a matching catch clause is located, the exception processing is completed by transferring control to the block of that catch clause. Within a matching catch clause, the variable `$_` contains a description of the current exception.
 - Otherwise, if the `try` block or a `catch` block of *S* encloses the throw point and if *S* has a `finally` block, control is transferred to the `finally` block. If the `finally` block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end of the `finally` block, processing of the current exception is continued.

- If an exception handler was not located in the current scope, the steps above are then repeated for the enclosing scope with a throw point corresponding to the statement from which the current scope was invoked.
- If the exception processing ends up terminating all scopes, indicating that no handler exists for the exception, then the behavior is unspecified.

To prevent unreachable catch clauses in a try block, a catch clause may not specify an exception type that is equal to or derived from a type that was specified in an earlier catch clause within that same try block.

The statements of a `finally` block are always executed when control leaves a `try` statement. This is true whether the control transfer occurs as a result of normal execution, as a result of executing a `break`, `continue`, or `return` statement, or as a result of an exception being thrown out of the `try` statement.

If an exception is thrown during execution of a `finally` block, the exception is thrown out to the next enclosing `try` statement. If another exception was in the process of being handled, that exception is lost. The process of generating an exception is further discussed in the description of the `throw` statement.

`try` statements can co-exist with `trap` statements; see §8.8 for details.

Examples:

```
PowerShell

$a = New-Object 'int[]' 10
$i = 20 # out-of-bounds subscript

while ($true) {
    try {
        $a[$i] = 10
        "Assignment completed without error"
        break
    }

    catch [IndexOutOfRangeException] {
        "Handling out-of-bounds index, >$_<`n"
        $i = 5
    }

    catch {
        "Caught unexpected exception"
    }

    finally {
        # ...
    }
}
```

```
    }  
}
```

Each exception thrown is raised as a `System.Management.Automation.RuntimeException`. If there are type-specific *catch-clauses* in the `try` block, the `InnerException` property of the exception is inspected to try and find a match, such as with the type `System.IndexOutOfRangeException` above.

8.8 The trap statement

Syntax:

Syntax

```
trap-statement:  
*trap* new-lines~opt~ type-literal~opt~ new-lines~opt~ statement-block
```

Description:

A `trap` statement with and without *type-literal* is analogous to a `catch` block ([§8.7](#)) with and without *catch-type-list*, respectively, except that a `trap` statement can trap only one type at a time.

Multiple `trap` statements can be defined in the same *statement-block*, and their order of definition is irrelevant. If two `trap` statements with the same *type-literal* are defined in the same scope, the lexically first one is used to process an exception of matching type.

Unlike a `catch` block, a `trap` statement matches an exception type exactly; no derived type matching is performed.

When an exception occurs, if no matching `trap` statement is present in the current scope, a matching trap statement is searched for in the enclosing scope, which may involve looking in the calling script, function, or filter, and then in its caller, and so on. If the lookup ends up terminating all scopes, indicating that no handler exists for the exception, then the behavior is unspecified.

A `trap` statement's *statement-body* only executes to process the corresponding exception; otherwise, execution passes over it.

If a `trap`'s *statement-body* exits normally, by default, an error object is written to the error stream, the exception is considered handled, and execution continues with the statement immediately following the one in the scope containing the `trap` statement

that made the exception visible. The cause of the exception might be in a command called by the command containing the `trap` statement.

If the final statement executed in a `trap`'s *statement-body* is `continue` ([§8.5.2](#)), the writing of the error object to the error stream is suppressed, and execution continues with the statement immediately following the one in the scope containing the `trap` statement that made the exception visible. If the final statement executed in a `trap`'s *statement-body* is `break` ([§8.5.1](#)), the writing of the error object to the error stream is suppressed, and the exception is re-thrown.

Within a `trap` statement the variable `$_` contains a description of the current error.

Consider the case in which an exception raised from within a `try` block does not have a matching `catch` block, but a matching `trap` statement exists at a higher block level. After the `try` block's `finally` clause is executed, the `trap` statement gets control even if any parent scope has a matching `catch` block. If a `trap` statement is defined within the `try` block itself, and that `try` block has a matching `catch` block, the `trap` statement gets control.

Examples:

In the following example, the error object is written and execution continues with the statement immediately following the one that caused the trap; that is, "Done" is written to the pipeline.

```
PowerShell  
  
$j = 0; $v = 10/$j; "Done"  
trap { $j = 2 }
```

In the following example, the writing of the error object is suppressed and execution continues with the statement immediately following the one that caused the trap; that is, "Done" is written to the pipeline.

```
PowerShell  
  
$j = 0; $v = 10/$j; "Done"  
trap { $j = 2; continue }
```

In the following example, the writing of the error object is suppressed and the exception is re-thrown.

```
PowerShell
```

```
$j = 0; $v = 10/$j; "Done"  
trap { $j = 2; break }
```

In the following example, the trap and exception-generating statements are in the same scope. After the exception is caught and handled, execution resumes with writing 1 to the pipeline.

PowerShell

```
&{trap{}; throw '\...'; 1}
```

In the following example, the trap and exception-generating statements are in different scopes. After the exception is caught and handled, execution resumes with writing 2 (not 1) to the pipeline.

PowerShell

```
trap{} &{throw '\...'; 1}; 2
```

8.9 The data statement

Syntax:

Syntax

```
data-statement:  
    data new-lines~opt~ data-name data-commands-allowed~opt~ statement-block  
  
data-name:  
    simple-name  
  
data-commands-allowed:  
    new-lines~opt~ -SupportedCommand data-commands-list  
  
data-commands-list:  
    new-lines~opt~ data-command  
    data-commands-list , new-lines~opt~ data-command  
  
data-command:  
    command-name-expr
```

Description:

A data statement creates a *data section*, keeping that section's data separate from the code. This separation supports facilities like separate string resource files for text, such as error messages and Help strings. It also helps support internationalization by making it easier to isolate, locate, and process strings that will be translated into different languages.

A script or function can have zero or more data sections.

The *statement-block* of a data section is limited to containing the following PowerShell features only:

- All operators except `-match`
- The `if` statement
- The following automatic variables: `$PSCulture`, `$PSUICulture`, `$true`, `$false`, and `$null`.
- Comments
- Pipelines
- Statements separated by semicolons (`;`)
- Literals
- Calls to the [ConvertFrom-StringData](#) cmdlet
- Any other cmdlets identified via the **SupportedCommand** parameter

If the `ConvertFrom-StringData` cmdlet is used, the key/value pairs can be expressed using any form of string literal. However, *expandable-string-literals* and *expandable-here-string-literals* must not contain any variable substitutions or sub-expression expansions.

Examples:

The **SupportedCommand** parameter indicates that the given cmdlets or functions generate data only. For example, the following data section includes a user-written cmdlet, `ConvertTo-Xml`, which formats data in an XML file:

PowerShell

```
data -SupportedCommand ConvertTo-Xml {
    Format-Xml -Strings string1, string2, string3
}
```

Consider the following example, in which the data section contains a `ConvertFrom-StringData` command that converts the strings into a hash table, whose value is assigned to `$messages`.

PowerShell

```
$messages = data {
    ConvertFrom-StringData -StringData @"
    Greeting = Hello
    Yes = yes
    No = no
'@
}
```

The keys and values of the hash table are accessed using `$messages.Greeting`, `$messages.Yes`, and `$messages.No`, respectively.

Now, this can be saved as an English-language resource. German- and Spanish-language resources can be created in separate files, with the following data sections:

PowerShell

```
$messages = data {
    ConvertFrom-StringData -StringData @@
    Greeting = Guten Tag
    Yes = ja
    No = nein
"@
}

$messagesS = data {
    ConvertFrom-StringData -StringData @@
    Greeting = Buenos días
    Yes = sí
    No = no
"@
}
```

If *dataname* is present, it names the variable (without using a leading \$) into which the value of the data statement is to be stored. Specifically, `$name = data { ... }` is equivalent to `data name { ... }`.

8.10 Function definitions

Syntax:

Syntax

```
function-statement:
    function new-lines~opt~ function-name function-parameter-
declaration~opt~ { script-block }
```

```

    filter new-lines~opt~ function-name function-parameter-declaration~opt~
{ script-block }
    workflow new-lines~opt~ function-name function-parameter-
declaration~opt~ { script-block }

function-name:
    command-argument

command-argument:
    command-name-expr

function-parameter-declaration:
    new-lines~opt~ ( parameter-list new-lines~opt~ )

parameter-list:
    script-parameter
    parameter-list new-lines~opt~ , script-parameter

script-parameter:
    new-lines~opt~ attribute-list~opt~ new-lines~opt~ variable script-
parameter-default~opt~

script-block:
    param-block~opt~ statement-terminators~opt~ script-block-body~opt~

param-block:
    new-lines~opt~ attribute-list~opt~ new-lines~opt~ param new-lines~opt~
    ( parameter-list~opt~ new-lines~opt~ )

parameter-list:
    script-parameter
    parameter-list new-lines~opt~ , script-parameter

script-parameter-default:
    new-lines~opt~ = new-lines~opt~ expression

script-block-body:
    named-block-list
    statement-list

named-block-list:
    named-block
    named-block-list named-block

named-block:
    block-name statement-block statement-terminators~opt~

block-name: one of
    dynamicparam    begin    process    end

```

Description:

A *function definition* specifies the name of the function, filter, or workflow being defined and the names of its parameters, if any. It also contains zero or more statements that are executed to achieve that function's purpose.

Each function is an instance of the class `System.Management.Automation.FunctionInfo`.

8.10.1 Filter functions

Whereas an ordinary function runs once in a pipeline and accesses the input collection via `$input`, a *filter* is a special kind of function that executes once for each object in the input collection. The object currently being processed is available via the variable `$_`.

A filter with no named blocks ([§8.10.7](#)) is equivalent to a function with a process block, but without any begin block or end block.

Consider the following filter function definition and calls:

```
PowerShell

filter Get-Square2 { # make the function a filter
    $_ * $_ # access current object from the collection
}

-3..3 | Get-Square2 # collection has 7 elements
6, 10, -3 | Get-Square2 # collection has 3 elements
```

Each filter is an instance of the class `System.Management.Automation.FilterInfo` ([§4.5.11](#)).

8.10.2 Workflow functions

A workflow function is like an ordinary function with implementation defined semantics. A workflow function is translated to a sequence of Windows Workflow Foundation activities and executed in the Windows Workflow Foundation engine.

8.10.3 Argument processing

Consider the following definition for a function called `Get-Power`:

```
PowerShell

function Get-Power ([long]$Base, [int]$Exponent) {
    $result = 1
    for ($i = 1; $i -le $Exponent; ++$i) {
        $result *= $Base
```

```
    }
    return $result
}
```

This function has two parameters, `$Base` and `$Exponent`. It also contains a set of statements that, for non-negative exponent values, computes `$Base^$Exponent^` and returns the result to `Get-Power`'s caller.

When a script, function, or filter begins execution, each parameter is initialized to its corresponding argument's value. If there is no corresponding argument and a default value ([§8.10.4](#)) is supplied, that value is used; otherwise, the value `$null` is used. As such, each parameter is a new variable just as if it was initialized by assignment at the start of the *script-block*.

If a *script-parameter* contains a type constraint (such as `[long]` and `[int]` above), the value of the corresponding argument is converted to that type, if necessary; otherwise, no conversion occurs.

When a script, function, or filter begins execution, variable `$args` is defined inside it as an unconstrained 1-dimensional array, which contains all arguments not bound by name or position, in lexical order.

Consider the following function definition and calls:

PowerShell

```
function F ($a, $b, $c, $d) { ... }

F -b 3 -d 5 2 4      # $a is 2, $b is 3, $c is 4, $d is 5, $args Length 0
F -a 2 -d 3 4 5      # $a is 2, $b is 4, $c is 5, $d is 3, $args Length 0
F 2 3 4 5 -c 7 -a 1  # $a is 1, $b is 2, $c is 7, $d is 3, $args Length 2
```

For more information about parameter binding see [§8.14](#).

8.10.4 Parameter initializers

The declaration of a parameter *p* may contain an initializer, in which case, that initializer's value is used to initialize *p* provided *p* is not bound to any arguments in the call.

Consider the following function definition and calls:

PowerShell

```
function Find-Str ([string]$Str, [int]$StartPos = 0) { ... }

Find-Str "abcabc" # 2nd argument omitted, 0 used for $StartPos
Find-Str "abcabc" 2 # 2nd argument present, so it is used for $StartPos
```

8.10.5 The [switch] type constraint

When a switch parameter is passed, the corresponding parameter in the command must be constrained by the type switch. Type switch has two values, True and False.

Consider the following function definition and calls:

PowerShell

```
function Process ([switch]$Trace, $P1, $P2) { ... }

Process 10 20          # $Trace is False, $P1 is 10, $P2 is 20
Process 10 -Trace 20    # $Trace is True, $P1 is 10, $P2 is 20
Process 10 20 -Trace    # $Trace is True, $P1 is 10, $P2 is 20
Process 10 20 -Trace:$false # $Trace is False, $P1 is 10, $P2 is 20
Process 10 20 -Trace:$true  # $Trace is True, $P1 is 10, $P2 is 20
```

8.10.6 Pipelines and functions

When a script, function, or filter is used in a pipeline, a collection of values is delivered to that script or function. The script, function, or filter gets access to that collection via the enumerator \$input ([\\$2.3.2.2](#), [\\$4.5.16](#)), which is defined on entry to that script, function, or filter.

Consider the following function definition and calls:

PowerShell

```
function Get-Square1 {
    foreach ($i in $input) {    # iterate over the collection
        $i * $i
    }
}

-3..3 | Get-Square1          # collection has 7 elements
6, 10, -3 | Get-Square1      # collection has 3 elements
```

8.10.7 Named blocks

The statements within a *script-block* can belong to one large unnamed block, or they can be distributed into one or more named blocks. Named blocks allow custom processing of collections coming from pipelines; named blocks can be defined in any order.

The statements in a *begin block* (i.e.; one marked with the keyword begin) are executed once, before the first pipeline object is delivered.

The statements in a *process block* (i.e.; one marked with the keyword process) are executed for each pipeline object delivered. (`$_` provides access to the current object being processed from the input collection coming from the pipeline.) This means that if a collection of zero elements is sent via the pipeline, the process block is not executed at all. However, if the script or function is called outside a pipeline context, this block is executed exactly once, and `$_` is set to `$null`, as there is no current collection object.

The statements in an *end block* (i.e.; one marked with the keyword end) are executed once, after the last pipeline object has been delivered.

8.10.8 dynamicparam block

The subsections of [§8.10](#) thus far deal with *static parameters*, which are defined as part of the source code. It is also possible to define *dynamic parameters* via a *dynamicparam block*, another form of named block ([§8.10.7](#)), which is marked with the keyword `dynamicparam`. Much of this machinery is implementation defined.

Dynamic parameters are parameters of a cmdlet, function, filter, or script that are available under certain conditions only. One such case is the **Encoding** parameter of the `Set-Item` cmdlet.

In the *statement-block*, use an if statement to specify the conditions under which the parameter is available in the function. Use the `New-Object` cmdlet to create an object of an implementation-defined type to represent the parameter, and specify its name. Also, use `New-Object` to create an object of a different implementation-defined type to represent the implementation-defined attributes of the parameter.

The following example shows a function with standard parameters called `Name` and `Path`, and an optional dynamic parameter named `DP1`. The `DP1` parameter is in the `PSet1` parameter set and has a type of `Int32`. The `DP1` parameter is available in the `Sample` function only when the value of the `Path` parameter contains "HKLM:", indicating that it is being used in the `HKEY_LOCAL_MACHINE` registry drive.

```

function Sample {
    param ([string]$Name, [string]$Path)
    dynamicparam {
        if ($Path -match "*HKLM*:") {
            $dynParam1 = New-Object
            System.Management.Automation.RuntimeDefinedParameter("dp1", [int32],
$attributeCollection)

            $attributes = New-Object
            System.Management.Automation.ParameterAttribute
                $attributes.ParameterSetName = 'pset1'
                $attributes.Mandatory = $false

                $attributeCollection = New-Object -Type
            System.Collections.ObjectModel.Collection`1[System.Attribute]
                $attributeCollection.Add($attributes)

                $paramDictionary = New-Object
            System.Management.Automation.RuntimeDefinedParameterDictionary
                $paramDictionary.Add("dp1", $dynParam1)
                return $paramDictionary
        }
    }
}

```

The type used to create an object to represent a dynamic parameter is

`System.Management.Automation.RuntimeDefinedParameter`.

The type used to create an object to represent the attributes of the parameter is

`System.Management.Automation.ParameterAttribute`.

The implementation-defined attributes of the parameter include **Mandatory**, **Position**, and **ValueFromPipeline**.

8.10.9 param block

A *param-block* provides an alternate way of declaring parameters. For example, the following sets of parameter declarations are equivalent:

PowerShell

```

function FindStr1 ([string]$Str, [int]$StartPos = 0) { ... }
function FindStr2 {
    param ([string]$Str, [int]$StartPos = 0) ...
}

```

A *param-block* allows an *attribute-list* on the *param-block* whereas a *function-parameter-declaration* does not.

A script may have a *param-block* but not a *function-parameter-declaration*. A function or filter definition may have a *function-parameter-declaration* or a *param-block*, but not both.

Consider the following example:

PowerShell

```
param ( [Parameter(Mandatory = $true, ValueFromPipeline=$true)]
      [string[]] $ComputerName )
```

The one parameter, `$ComputerName`, has type `string[]`, it is required, and it takes input from the pipeline.

See [§12.3.7](#) for a discussion of the **Parameter** attribute and for more examples.

8.11 The parallel statement

Syntax:

Syntax

```
parallel-statement:
  *parallel* statement-block
```

The parallel statement contains zero or more statements that are executed in an implementation defined manner.

A parallel statement is only allowed in a workflow ([§8.10.2](#)).

8.12 The sequence statement

Syntax:

Syntax

```
sequence-statement:
  *sequence* statement-block
```

The sequence statement contains zero or more statements that are executed in an implementation defined manner.

A sequence statement is only allowed in a workflow ([§8.10.2](#)).

8.13 The inlinescript statement

Syntax:

```
Syntax  
  
inlinescript-statement:  
    inlinescript statement-block
```

The inlinescript statement contains zero or more statements that are executed in an implementation defined manner.

An inlinescript statement is only allowed in a workflow ([§8.10.2](#)).

8.14 Parameter binding

When a script, function, filter, or cmdlet is invoked, each argument can be bound to the corresponding parameter by position, with the first parameter having position zero.

Consider the following definition fragment for a function called `Get-Power`, and the calls to it:

```
PowerShell  
  
function Get-Power ([long]$Base, [int]$Exponent) { ... }  
  
Get-Power 5 3      # argument 5 is bound to parameter $Base in position 0  
                  # argument 3 is bound to parameter $Exponent in position  
1  
                  # no conversion is needed, and the result is 5 to the  
power 3  
  
Get-Power 4.7 3.2  # double argument 4.7 is rounded to int 5, double  
argument          # 3.2 is rounded to int 3, and result is 5 to the power  
3  
  
Get-Power 5        # $Exponent has value $null, which is converted to int 0  
  
Get-Power          # both parameters have value $null, which is converted  
to int 0
```

When a script, function, filter, or cmdlet is invoked, an argument can be bound to the corresponding parameter by name. This is done by using a *parameter with argument*, which is an argument that is the parameter's name with a leading dash (-), followed by the associated value for that argument. The parameter name used can have any case-insensitive spelling and can use any prefix that uniquely designates the corresponding parameter. When choosing parameter names, avoid using the names of the [common parameters](#).

Consider the following calls to function `Get-Power`:

```
PowerShell

Get-Power -Base 5 -Exponent 3 # -Base designates $Base, so 5 is
                                # bound to that, -Exponent designates
                                # $Exponent, so 3 is bound to that

Get-Power -Exp 3 -Bas 5          # $Base takes on 5 and $Exponent takes on 3

Get-Power -E 3 -B 5            # $Base takes on 5 and $Exponent takes on 3
```

On the other hand, calls to the following function

```
PowerShell

function Get-Hypot ([double]$Side1, [double]$Side2) {
    return [Math]::Sqrt($Side1 * $Side1 + $Side2 * $Side2)
}
```

must use parameters `-Side1` and `-Side2`, as there is no prefix that uniquely designates the parameter.

The same parameter name cannot be used multiple times with or without different associated argument values.

Parameters can have attributes (§12). For information about the individual attributes see the sections within [§12.3](#). For information about parameter sets see [§12.3.7](#).

A script, function, filter, or cmdlet can receive arguments via the invocation command line, from the pipeline, or from both. Here are the steps, in order, for resolving parameter binding:

1. Bind all named parameters, then
2. Bind positional parameters, then
3. Bind from the pipeline by value ([§12.3.7](#)) with exact match, then
4. Bind from the pipeline by value ([§12.3.7](#)) with conversion, then

5. Bind from the pipeline by name ([§12.3.7](#)) with exact match, then
6. Bind from the pipeline by name ([§12.3.7](#)) with conversion

Several of these steps involve conversion, as described in [§6](#). However, the set of conversions used in binding is not exactly the same as that used in language conversions. Specifically,

- Although the value `$null` can be cast to `bool`, `$null` cannot be bound to `bool`.
- When the value `$null` is passed to a switch parameter for a cmdlet, it is treated as if `$true` was passed. However, when passed to a switch parameter for a function, it is treated as if `$false` was passed.
- Parameters of type `bool` or `switch` can only bind to numeric or `bool` arguments.
- If the parameter type is not a collection, but the argument is some sort of collection, no conversion is attempted unless the parameter type is `object` or `PsObject`. (The main point of this restriction is to disallow converting a collection to a string parameter.) Otherwise, the usual conversions are attempted.

If the parameter type is `IList` or `ICollection<T>`, only those conversions via `Constructor`, `op_Implicit`, and `op_Explicit` are attempted. If no such conversions exist, a special conversion for parameters of "collection" type is used, which includes `IList`, `ICollection<T>`, and arrays.

Positional parameters prefer to be bound without type conversion, if possible. For example,

PowerShell

```
function Test {
    [CmdletBinding(DefaultParameterSetName = "SetB")]
    param([Parameter(Position = 0, ParameterSetName = "SetA")]
        [decimal]$Dec,
        [Parameter(Position = 0, ParameterSetName = "SetB")]
        [int]$In
    )
    $PSCmdlet.ParameterSetName
}

Test 42d # outputs "SetA"
Test 42 # outputs "SetB"
```

9. Arrays

Article • 01/08/2025

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

9.1 Introduction

PowerShell supports arrays of one or more dimensions with each dimension having zero or more *elements*. Within a dimension, elements are numbered in ascending integer order starting at zero. Any individual element can be accessed via the array subscript operator `[]` ([§7.1.4](#)). The number of dimensions in an array is called its *rank*.

An element can contain a value of any type including an array type. An array having one or more elements whose values are of any array type is called a *jagged array*. A *multidimensional array* has multiple dimensions, in which case, the number of elements in each row of a dimension is the same. An element of a jagged array may contain a multidimensional array, and vice versa.

Multidimensional arrays are stored in row-major order. The number of elements in an array is called that array's *length*, which is fixed when the array is created. As such, the elements in a 1-dimensional array *A* having length *N* can be accessed (i.e., *subscripted*) using the expressions `A[0], A[1], ..., A[N-1]`. The elements in a 2-dimensional array *B* having *M* rows, with each row having *N* columns, can be accessed using the expressions

`B[0,0], B[0,1], ..., B[0,N-1], B[1,0], B[1,1], ..., B[1,N-1], ..., B[M-1,0], B[M-1,1], ..., B[M-1,N-1]`. And so on for arrays with three or more dimensions.

By default, an array is *polymorphic*; i.e., its elements do not need to all have the same type. For example,

PowerShell

```
$items = 10, "blue", 12.54e3, 16.30D # 1-D array of length 4
$items[1] = -2.345
$items[2] = "green"

$a = New-Object 'object[,]' 2,2 # 2-D array of length 4
$a[0,0] = 10
$a[0,1] = $false
$a[1,0] = "red"
$a[1,1] = $null
```

A 1-dimensional array has type `type[]`, a 2-dimensional array has type `type[,]`, a 3-dimensional array has type `type[,,]`, and so on, where `type` is `object` for an unconstrained type array, or the constrained type for a constrained array ([§9.4](#)).

All array types are derived from the type `Array` ([§4.3.2](#)).

9.2 Array creation

An array is created via an *array creation expression*, which has the following forms: unary comma operator ([§7.2.1](#)), *array-expression* ([§7.1.7](#)), binary comma operator ([§7.3](#)), range operator ([§7.4](#)), or `New-Object` cmdlet.

Here are some examples of array creation and usage:

PowerShell

```
$values = 10, 20, 30
for ($i = 0; $i -lt $values.Length; ++$i) {
    "$values[$i] = $($values[$i])"
}

$x = , 10                                # x refers to an array of length 1
$x = @(10)                                 # x refers to an array of length 1
$x = @()                                    # x refers to an array of length 0

$a = New-Object 'object[,]' 2, 2 # create a 2x2 array of anything
$a[0, 0] = 10                            # set to an int value
$a[0, 1] = $false                          # set to a boolean value
$a[1, 0] = "red"                           # set to a string value
$a[1, 1] = 10.50D                         # set to a decimal value
```

```
foreach ($e in $a) {                      # enumerate over the whole array
    $e
}
```

The following is written to the pipeline:

Output

```
$values[0] = 10
$values[1] = 20
$values[2] = 30

10
False
red
10.50
```

The default initial value of any element not explicitly initialized is the default value for that element's type (that is, `$false`, zero, or `$null`).

9.3 Array concatenation

Arrays of arbitrary type and length can be concatenated via the `+` and `+=` operators, both of which result in the creation of a new unconstrained 1-dimensional array. The existing arrays are unchanged. See [§7.7.3](#) for more information, and [§9.4](#) for a discussion of adding to an array of constrained type.

9.4 Constraining element types

A 1-dimensional array can be created so that it is type-constrained by prefixing the array-creation expression with an array type cast. For example,

PowerShell

```
$a = [int[]](1,2,3,4)      # constrained to int
$a[1] = "abc"              # implementation-defined behavior
$a += 1.23                 # new array is unconstrained
```

The syntax for creating a multidimensional array requires the specification of a type, and that type becomes the constraint type for that array. However, by specifying type `object[]`, there really is no constraint as a value of any type can be assigned to an element of an array of that type.

Concatenating two arrays ([§7.7.3](#)) always results in a new array that is unconstrained even if both arrays are constrained by the same type. For example,

PowerShell

```
$a = [int[]](1,2,3)      # constrained to int
$b = [int[]](10,20)      # constrained to int
$c = $a + $b            # constraint not preserved
$c = [int[]]($a + $b)    # result explicitly constrained to int
```

9.5 Arrays as reference types

As array types are reference types, a variable designating an array can be made to refer to any array of any rank, length, and element type. For example,

PowerShell

```
$a = 10,20                  # $a refers to an array of length 2
$a = 10,20,30                # $a refers to a different array, of length 3
$a = "red",10.6              # $a refers to a different array, of length 2
$a = New-Object 'int[,]' 2,3  # $a refers to an array of rank 2
```

Assignment of an array involves a shallow copy; that is, the variable assigned to refers to the same array, no copy of the array is made. For example,

PowerShell

```
$a = 10,20,30
">$a<"                   # $a is a reference to an array of length 3
$b = $a                     # make $b refer to the same array as $a
">$b<"                   # $b is a reference to the same array as $a

$a[0] = 6                  # change value of [0] via $a
">$a<"                   # $a is still a reference to the same array
">$b<"                   # change is reflected in $b

$b += 40                  # make $b refer to a new array
$a[0] = 8                  # change value of [0] via $a
">$a<"                   # $a is still a reference to the same array
">$b<"                   # change is not reflected in $b
```

The following is written to the pipeline:

Output

```
>10 20 30<
>10 20 30<
```

```
>6 20 30<
>6 20 30<
>8 20 30<
>6 20 30 40<
```

9.6 Arrays as array elements

Any element of an array can itself be an array. For example,

PowerShell

```
$colors = "red", "blue", "green"
$list = $colors, (,7), (1.2, "yes") # parens in (,7) are redundant; they
                                    # are intended to aid readability
``$list refers to an array of length $($list.Length)``
">$($list[1][0])<"
```

The following is written to the pipeline:

Output

```
$list refers to an array of length 3
>7<
>yes<
```

`$list[1]` refers to an array of 1 element, the integer 7, which is accessed via `$list[1][0]`, as shown. Compare this with the following subtly different case:

PowerShell

```
$list = $colors, 7, (1.2, "yes") # 7 has no prefix comma
">$($list[1])<"
```

Here, `$list[1]` refers to a scalar, the integer 7, which is accessed via `$list[1]`.

Consider the following example,

PowerShell

```
$x = [string[]]("red", "green")
$y = 12.5, $true, "blue"
$a = New-Object 'object[,]' 2,2
$a[0,0] = $x                      # element is an array of 2 strings
$a[0,1] = 20                        # element is an int
```

```
$a[1,0] = $y          # element is an array of 3 objects
$a[1,1] = [int[]](92,93)  # element is an array of 2 ints
```

9.7 Negative subscripting

This is discussed in [§7.1.4.1](#).

9.8 Bounds checking

This is discussed in [§7.1.4.1](#).

9.9 Array slices

An *array slice* is an unconstrained 1-dimensional array whose elements are copies of zero or more elements from a collection. An array slice is created via the subscript operator `[]` ([§7.1.4.5](#)).

9.10 Copying an array

A contiguous set of elements can be copied from one array to another using the method `[array]::Copy`. For example,

PowerShell

```
$a = [int[]](10,20,30)
$b = [int[]](0,1,2,3,4,5)
[array]::Copy($a, $b, 2)      # $a[0]->$b[0],
$a[1]->$b[1]
[array]::Copy($a, 1, $b, 3, 2) # $a[1]->$b[3],
$a[2]->$b[4]
```

9.11 Enumerating over an array

Although it is possible to loop through an array accessing each of its elements via the subscript operator, we can enumerate over that array's elements using the `foreach` statement. For a multidimensional array, the elements are processed in row-major order. For example,

PowerShell

```

$a = 10, 53, 16, -43
foreach ($elem in $a) {
    # do something with element via $elem
}

foreach ($elem in -5..5) {
    # do something with element via $elem
}

$a = New-Object 'int[,]' 3, 2
foreach ($elem in $a) {
    # do something with element via $elem
}

```

9.12 Multidimensional array flattening

Some operations on a multidimensional array (such as replication ([§7.6.3](#)) and concatenation ([§7.7.3](#))) require that array to be *flattened*; that is, to be turned into a 1-dimensional array of unconstrained type. The resulting array takes on all the elements in row-major order.

Consider the following example:

PowerShell

```

$a = "red",$true
$b = (New-Object 'int[,]' 2,2)
$b[0,0] = 10
$b[0,1] = 20
$b[1,0] = 30
$b[1,1] = 40
$c = $a + $b

```

The array designated by `$c` contains the elements "red", `$true`, 10, 20, 30, and 40.

10. Hashtables

Article • 01/08/2025

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

Syntax:

Tip

The `~opt~` notation in the syntax definitions indicates that the lexical entity is optional in the syntax.

Syntax

```
hash-literal-expression:  
    @{ new-lines~opt~ hash-literal-body~opt~ new-lines~opt~ }  
  
hash-literal-body:  
    hash-entry  
    hash-literal-body statement-terminators hash-entry  
  
hash-entry:  
    key-expression = new-lines~opt~ statement  
  
key-expression:  
    simple-name  
    unary-expression
```

```
statement-terminator:  
;  
new-line-character
```

10.1 Introduction

The type `Hashtable` represents a collection of *key/value pair* objects that supports efficient retrieval of a value when indexed by the key. Each key/value pair is an *element*, which is stored in some implementation-defined object type.

An element's key cannot be the null value. There are no restrictions on the type of a key or value. Duplicate keys are not supported.

Given a key/value pair object, the key and associated value can be obtained by using the instance properties `Key` and `Value`, respectively.

Given one or more keys, the corresponding value(s) can be accessed via the `Hashtable` subscript operator `[]` ([§7.1.4.3](#)).

All `Hashtables` have type `Hashtable` ([§4.3.3](#)).

The order of the keys in the collection returned by `Keys` is unspecified; however, it is the same order as the associated values in the collection returned by `Values`.

Here are some examples involving `Hashtables`:

PowerShell

```
$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }  
$h1.FirstName # designates the key FirstName  
$h1["LastName"] # designates the associated value for key LastName  
$h1.Keys # gets the collection of keys
```

`Hashtable` elements are stored in an object of type `DictionaryEntry`, and the collections returned by `Keys` and `Values` have type `ICollection`.

10.2 Hashtable creation

A `Hashtable` is created via a hash literal ([§7.1.9](#)) or the `New-Object` cmdlet. It can be created with zero or more elements. The `Count` property returns the current element count.

10.3 Adding and removing Hashtable elements

An element can be added to a `Hashtable` by assigning ([\\$7.11.1](#)) a value to a non-existent key name or to a subscript ([\\$7.1.4.3](#)) that uses a non-existent key name. Removal of an element requires the use of the `Remove` method. For example,

PowerShell

```
$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }
$h1.Dept = "Finance" # adds element Finance
$h1["Salaried"] = $false # adds element Salaried
$h1.Remove("Salaried") # removes element Salaried
```

10.4 Hashtable concatenation

Hashtables can be concatenated via the `+` and `+=` operators, both of which result in the creation of a new `Hashtable`. The existing Hashtables are unchanged. See [\\$7.7.4](#) for more information.

10.5 Hashtables as reference types

As `Hashtable` is a reference type, assignment of a `Hashtable` involves a shallow copy; that is, the variable assigned to refers to the same `Hashtable`; no copy of the `Hashtable` is made. For example,

PowerShell

```
$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123 }
$h2 = $h1
$h1.FirstName = "John" # change key's value in $h1
$h2.FirstName # change is reflected in $h2
```

10.6 Enumerating over a Hashtable

To process every pair in a `Hashtable`, use the `Keys` property to retrieve the list of keys as an array, and then enumerate over the elements of that array getting the associated value via the `Value` property or a subscript, as follows

PowerShell

```
$h1 = @{ FirstName = "James"; LastName = "Anderson"; IDNum = 123}
foreach ($e in $h1.Keys) {
    "Key is " + $e + ", Value is " + $h1[$e]
}
```

11. Modules

Article • 01/08/2025

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

11.1 Introduction

As stated in §3.14, a module is a self-contained reusable unit that allows PowerShell code to be partitioned, organized, and abstracted. A module can contain one or more *module members*, which are commands (such as cmdlets and functions) and items (such as variables and aliases). The names of these members can be kept private to the module or they may be *exported* to the session into which the module is *imported*.

There are three different *module types*: manifest, script, and binary. A *manifest module* is a file that contains information about a module, and controls certain aspects of that module's use. A *script module* is a PowerShell script file with a file extension of `.psm1` instead of `.ps1`. A *binary module* contains class types that define cmdlets and providers. Unlike script modules, binary modules are written in compiled languages. Binary modules are not covered by this specification.

A binary module is a .NET assembly (i.e.; a DLL) that was compiled against the PowerShell libraries.

Modules may *nest*; that is, one module may import another module. A module that has associated nested modules is a *root module*.

When a PowerShell session is created, by default, no modules are imported.

When modules are imported, the search path used to locate them is defined by the environment variable **PSModulePath**.

The following cmdlets deal with modules:

- [Get-Module](#): Identifies the modules that have been, or can be imported
- [Import-Module](#): Adds one or more modules to the current session (see [§11.4](#))
- [Export-ModuleMember](#): Identifies the module members that are to be exported
- [Remove-Module](#): Removes one or more modules from the current session (see [§11.5](#))
- [New-Module](#): Creates a dynamic module (see [§11.7](#))

11.2 Writing a script module

A script module is a script file. Consider the following script module:

```
PowerShell

function Convert-CentigradeToFahrenheit ([double]$tempC) {
    return ($tempC * (9.0 / 5.0)) + 32.0
}
New-Alias c2f Convert-CentigradeToFahrenheit

function Convert-FahrenheitToCentigrade ([double]$tempF) {
    return ($tempF - 32.0) * (5.0 / 9.0)
}
New-Alias f2c Convert-FahrenheitToCentigrade

Export-ModuleMember -Function Convert-CentigradeToFahrenheit
Export-ModuleMember -Function Convert-FahrenheitToCentigrade
Export-ModuleMember -Alias c2f, f2c
```

This module contains two functions, each of which has an alias. By default, all function names, and only function names are exported. However, once the cmdlet `Export-ModuleMember` has been used to export anything, then only those things exported explicitly will be exported. A series of commands and items can be exported in one call or a number of calls to this cmdlet; such calls are cumulative for the current session.

11.3 Installing a script module

A script module is defined in a script file, and modules can be stored in any directory. The environment variable PSModulePath points to a set of directories to be searched when module-related cmdlets look for modules whose names do not include a fully qualified path. Additional lookup paths can be provided; for example,

```
$Env:PSModulePath = $Env:PSModulePath + ";<additional-path>"
```

Any additional paths added affect the current session only.

Alternatively, a fully qualified path can be specified when a module is imported.

11.4 Importing a script module

Before the resources in a module can be used, that module must be imported into the current session, using the cmdlet `Import-Module`. `Import-Module` can restrict the resources that it actually imports.

When a module is imported, its script file is executed. That process can be configured by defining one or more parameters in the script file, and passing in corresponding arguments via the `ArgumentList` parameter of `Import-Module`.

Consider the following script that uses these functions and aliases defined in §11.2:

```
Import-Module "E:\Scripts\Modules\PSTest_Temperature" -Verbose
```

PowerShell

```
"0 degrees C is " + (Convert-CentigradeToFahrenheit 0) + " degrees F"
"100 degrees C is " + (c2f 100) + " degrees F"
"32 degrees F is " + (Convert-FahrenheitToCentigrade 32) + " degrees C"
"212 degrees F is " + (f2c 212) + " degrees C"
```

Importing a module causes a name conflict when commands or items in the module have the same names as commands or items in the session. A name conflict results in a name being hidden or replaced. The `Prefix` parameter of `Import-Module` can be used to avoid naming conflicts. Also, the `Alias`, `Cmdlet`, `Function`, and `Variable` parameters can limit the selection of commands to be imported, thereby reducing the chances of name conflict.

Even if a command is hidden, it can be run by qualifying its name with the name of the module in which it originated. For example, `& M\F 100` invokes the function `F` in module `M`, and passes it the argument 100.

When the session includes commands of the same kind with the same name, such as two cmdlets with the same name, by default it runs the most recently added command.

See [§3.5.6](#) for a discussion of scope as it relates to modules.

11.5 Removing a script module

One or more modules can be removed from a session via the cmdlet `Remove-Module`.

Removing a module does not uninstall the module.

In a script module, it is possible to specify code that is to be executed prior to that module's removal, as follows:

```
$MyInvocation.MyCommand.ScriptBlock.Module.OnRemove = { *on-removal-code* }
```

11.6 Module manifests

As stated in [§11.1](#), a manifest module is a file that contains information about a module, and controls certain aspects of that module's use.

A module need not have a corresponding manifest, but if it does, that manifest has the same name as the module it describes, but with a `.psd1` file extension.

A manifest contains a limited subset of PowerShell script, which returns a Hashtable containing a set of keys. These keys and their values specify the *manifest elements* for that module. That is, they describe the contents and attributes of the module, define any prerequisites, and determine how the components are processed.

Essentially, a manifest is a data file; however, it can contain references to data types, the if statement, and the arithmetic and comparison operators. (Assignments, function definitions and loops are not permitted.) A manifest also has read access to environment variables and it can contain calls to the cmdlet `Join-Path`, so paths can be constructed.

ⓘ Note

Editor's Note: The original document contains a list of keys allowed in a module manifest file. That list is outdated and incomplete. For a complete list of keys in a module manifest, see [New-ModuleManifest](#).

The only key that is required is `ModuleVersion`.

Here is an example of a simple manifest:

PowerShell

```
@{
    ModuleVersion = '1.0'
    Author = 'John Doe'
    RequiredModules = @()
    FunctionsToExport = 'Set*', 'Get*', 'Process*'
}
```

The key **GUID** has a `string` value. This specifies a Globally Unique Identifier (GUID) for the module. The **GUID** can be used to distinguish among modules having the same name. To create a new GUID, call the method `[guid]::NewGuid()`.

11.7 Dynamic modules

A *dynamic module* is a module that is created in memory at runtime by the cmdlet `New-Module`; it is not loaded from disk. Consider the following example:

PowerShell

```
$sb = {
    function Convert-CentigradeToFahrenheit ([double]$tempC) {
        return ($tempC * (9.0 / 5.0)) + 32.0
    }

    New-Alias c2f Convert-CentigradeToFahrenheit

    function Convert-FahrenheitToCentigrade ([double]$tempF) {
        return ($tempF - 32.0) * (5.0 / 9.0)
    }

    New-Alias f2c Convert-FahrenheitToCentigrade

    Export-ModuleMember -Function Convert-CentigradeToFahrenheit
    Export-ModuleMember -Function Convert-FahrenheitToCentigrade
    Export-ModuleMember -Alias c2f, f2c
}

New-Module -Name MyDynMod -ScriptBlock $sb
Convert-CentigradeToFahrenheit 100
c2f 100
```

The script block `$sb` defines the contents of the module, in this case, two functions and two aliases to those functions. As with an on-disk module, only functions are exported

by default, so `Export-ModuleMember` cmdlets calls exist to export both the functions and the aliases.

Once `New-Module` runs, the four names exported are available for use in the session, as is shown by the calls to the `Convert-CentigradeToFahrenheit` and `c2f`.

Like all modules, the members of dynamic modules run in a private module scope that is a child of the global scope. `Get-Module` cannot get a dynamic module, but `Get-Command` can get the exported members.

To make a dynamic module available to `Get-Module`, pipe a `New-Module` command to `Import-Module`, or pipe the module object that `New-Module` returns, to `Import-Module`. This action adds the dynamic module to the `Get-Module` list, but it does not save the module to disk or make it persistent.

11.8 Closures

A dynamic module can be used to create a *closure*, a function with attached data. Consider the following example:

PowerShell

```
function Get-NextID ([int]$StartValue = 1) {
    $nextID = $StartValue
    {
        ($Script:nextID++)
    }.GetNewClosure()
}

$v1 = Get-NextID      # get a scriptblock with $StartValue of 0
& $v1                 # invoke Get-NextID getting back 1
& $v1                 # invoke Get-NextID getting back 2

$v2 = Get-NextID 100  # get a scriptblock with $StartValue of 100
& $v2                 # invoke Get-NextID getting back 100
& $v2                 # invoke Get-NextID getting back 101
```

The intent here is that `Get-NextID` return the next ID in a sequence whose start value can be specified. However, multiple sequences must be supported, each with its own `$StartValue` and `$nextID` context. This is achieved by the call to the method `[scriptblock]::GetNewClosure` ([\\$4.3.7](#)).

Each time a new closure is created by `GetNewClosure`, a new dynamic module is created, and the variables in the caller's scope (in this case, the script block containing the increment) are copied into this new module. To ensure that the `nextId` defined inside the

parent function (but outside the script block) is incremented, the explicit Script: scope prefix is needed.

Of course, the script block need not be a named function; for example:

PowerShell

```
$v3 = & {      # get a scriptblock with $StartValue of 200
    param ([int]$StartValue = 1)
    $nextID = $StartValue
    {
        ($Script:nextID++)
        }.GetNewClosure()
} 200

& $v3          # invoke script getting back 200
& $v3          # invoke script getting back 201
```

12. Attributes

Article • 03/24/2025

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

An *attribute* object associates predefined system information with a *target element*, which can be a param block or a parameter ([§8.10](#)). Each attribute object has an *attribute type*.

Information provided by an attribute is also known as *metadata*. Metadata can be examined by the command or the execution environment to control how the command processes data or before run time by external tools to control how the command itself is processed or maintained.

Multiple attributes can be applied to the same target element.

12.1 Attribute specification

Tip

The `~opt~` notation in the syntax definitions indicates that the lexical entity is optional in the syntax.

Syntax

```
attribute-list:  
    attribute  
    attribute-list new-lines~opt~ attribute  
  
attribute:  
    [ new-lines~opt~ attribute-name ( attribute-arguments new-lines~opt~ )  
new-lines~opt~ ]  
    type-literal  
  
attribute-name:  
    type-spec  
  
attribute-arguments:  
    attribute-argument  
    attribute-argument new-lines~opt~,  
    attribute-arguments  
  
attribute-argument:  
    new-lines~opt~ expression  
    new-lines~opt~ simple-name  
    new-lines~opt~ simple-name = new-lines~opt~ expression
```

An attribute consists of an *attribute-name* and an optional list of positional and named arguments. The positional arguments (if any) precede the named arguments. A named argument consists of a *simple-name*, optionally followed by an equal sign and followed by an *expression*. If the expression is omitted, the value `$true` is assumed.

The *attribute-name* is a reserved attribute type ([§12.3](#)) or some implementation-defined attribute type.

12.2 Attribute instances

An attribute instance is an object of an attribute type. The instance represents an attribute at run-time.

To create an object of some attribute type *A*, use the notation `A()`. An attribute is declared by enclosing its instance inside `[]`, as in `[A()]`. Some attribute types have positional and named parameters ([§8.14](#)), just like functions and cmdlets. For example,

```
[A(10,IgnoreCase=$true)]
```

shows an instance of type *A* being created using a positional parameter whose argument value is 10, and a named parameter, `IgnoreCase`, whose argument value is `$true`.

12.3 Reserved attributes

The attributes described in the following sections can be used to augment or modify the behavior of PowerShell functions, filters, scripts, and cmdlets.

12.3.1 The Alias attribute

This attribute is used in a *script-parameter* to specify an alternate name for a parameter. A parameter may have multiple aliases, and each alias name must be unique within a *parameter-list*. One possible use is to have different names for a parameter in different parameter sets (see **ParameterSetName**).

The attribute argument has type `string[]`.

Consider a function call `Test1` that has the following param block, and which is called as shown:

```
PowerShell

param (
    [Parameter(Mandatory = $true)]
    [Alias("CN")]
    [Alias("Name", "System")]
    [string[]] $ComputerName
)

Test1 "Mars", "Saturn"          # pass argument by position
Test1 -ComputerName "Mars", "Saturn" # pass argument by name
Test1 -CN "Mars", "Saturn"       # pass argument using first alias
Test1 -Name "Mars", "Saturn"     # pass argument using second alias
Test1 -Sys "Mars", "Saturn"      # pass argument using third alias
```

Consider a function call `Test2` that has the following param block, and which is called as shown:

```
PowerShell

param (
    [Parameter(Mandatory = $true, ValueFromPipelineByPropertyName = $true)]
    [Alias('PSPPath')]
    [string] $LiteralPath
)

Get-ChildItem "E:\*.txt" | Test2 -LiteralPath { $_ ; "`n`t";
    $_.FullName + ".bak" }
Get-ChildItem "E:\*.txt" | Test2
```

Cmdlet `Get-ChildItem` (alias `dir`) adds to the object it returns a new `NoteProperty` of type `string`, called `PSPPath`.

12.3.2 The AllowEmptyCollection attribute

This attribute is used in a *script-parameter* to allow an empty collection as the argument of a mandatory parameter.

Consider a function call `Test` that has the following param block, and which is called as shown:

```
PowerShell

param (
    [Parameter(Mandatory = $true)]
    [AllowEmptyCollection()]
    [string[]] $ComputerName
)

Test "Red", "Green" # $ComputerName has Length 2
Test "Red" # $ComputerName has Length 1
Test -Comp @() # $ComputerName has Length 0
```

12.3.3 The AllowEmptyString attribute

This attribute is used in a *script-parameter* to allow an empty string as the argument of a mandatory parameter.

Consider a function call `Test` that has the following param block, and which is called as shown:

```
PowerShell

param (
    [Parameter(Mandatory = $true)]
    [AllowEmptyString()]
    [string] $ComputerName
)

Test "Red" # $ComputerName is "Red"
Test "" # empty string is permitted
Test -Comp "" # empty string is permitted
```

12.3.4 The AllowNull attribute

This attribute is used in a *script-parameter* to allow `$null` as the argument of a mandatory parameter for which no implicit conversion is available.

Consider a function call `Test` that has the following param block, and which is called as shown:

```
PowerShell

param (
    [Parameter(Mandatory = $true)]
    [AllowNull()]
    [int[]] $Values
)

Test 10, 20, 30      # $values has Length 3, values 10, 20, 30
Test 10, $null, 30   # $values has Length 3, values 10, 0, 30
Test -Val $null       # $values has value $null
```

Note that the second case above does not need this attribute; there is already an implicit conversion from `$null` to int.

12.3.5 The `CmdletBinding` attribute

This attribute is used in the *attribute-list* of *param-block* of a function to indicate that function acts similar to a cmdlet. Specifically, it allows functions to access a number of methods and properties through the `$PSCmdlet` variable by using begin, process, and end named blocks ([§8.10.7](#)).

When this attribute is present, positional arguments that have no matching positional parameters cause parameter binding to fail and `$args` is not defined. (Without this attribute `$args` would take on any unmatched positional argument values.)

The following arguments are used to define the characteristics of the parameter:

[] Expand table

Parameter Name	Purpose
SupportsShouldProcess (named)	Type: bool; Default value: \$false Specifies whether the function supports calls to the <code>ShouldProcess</code> method, which is used to prompt the user for feedback before the function makes a change to the system. A value of <code>\$true</code> indicates that it does. A value of <code>\$false</code> indicates that it doesn't.
ConfirmImpact (named)	Type: string; Default value: "Medium"

Parameter Name	Purpose
	<p>Specifies the impact level of the action performed. The call to the ShouldProcess method displays a confirmation prompt only when the ConfirmImpact argument is greater than or equal to the value of the \$ConfirmPreference preference variable.</p> <p>The possible values of this argument are:</p> <ul style="list-style-type: none"> None: Suppress all requests for confirmation. Low: The action performed has a low risk of losing data. Medium: The action performed has a medium risk of losing data. High: The action performed has a high risk of losing data. <p>The value of \$ConfirmPreference can be set so that only cmdlets with an equal or higher impact level can request confirmation before they perform their operation. For example, if \$ConfirmPreference is set to Medium, cmdlets with a Medium or High impact level can request confirmation. Requests from cmdlets with a low impact level are suppressed.</p>
DefaultParameterSetName (named)	<p>Type: string; Default value: "__AllParameterSets"</p> <p>Specifies the parameter set to use if that cannot be determined from the arguments. See the named argument ParameterSetName in the attribute Parameter ([§12.3.7][§12.3.7]).</p>
PositionalBinding (named)	<p>Type: bool; Default value: \$true</p> <p>Specifies whether positional binding is supported or not. The value of this argument is ignored if any parameters specify non-default values for either the named argument Position or the named argument ParameterSetName in the attribute Parameter ([§12.3.7][§12.3.7]). Otherwise, if the argument is \$false then no parameters are positional, otherwise parameters are assigned a position based on the order the parameters are specified.</p>

Here's is an example of the framework for using this attribute:

```

PowerShell

[CmdletBinding(SupportsShouldProcess = $true, ConfirmImpact = "Low")]
param ( ... )

begin { ... }
Get-process { ... }
end { ... }

```

12.3.6 The OutputType attribute

This attribute is used in the *attribute-list* of *param-block* to specify the types returned. The following arguments are used to define the characteristics of the parameter:

[+] Expand table

Parameter Name	Purpose
Type (position 0)	Type: string[] or array of type literals A list of the types of the values that are returned.
ParameterSetName (named)	Type: string[] Specifies the parameter sets that return the types indicated by the corresponding elements of the Type parameter.

Here are several examples of this attribute's use:

PowerShell

```
[OutputType([int])] param ( ... )
[OutputType("double")] param ( ... )
[OutputType("string","string")] param ( ... )
```

12.3.7 The Parameter attribute

This attribute is used in a *script-parameter*. The following named arguments are used to define the characteristics of the parameter:

[+] Expand table

Parameter	Purpose
HelpMessage (named)	Type: string This argument specifies a message that is intended to contain a short description of the parameter. This message is used in an implementation-defined manner when the function or cmdlet is run yet a mandatory parameter having a HelpMessage does not have a corresponding argument. The following example shows a parameter declaration that provides a description of the parameter.

Parameter	Purpose
	<pre data-bbox="653 159 1257 278">param ([Parameter(Mandatory = \$true, HelpMessage = "An array of computer names.")] [string[]] \$ComputerName)</pre> <p data-bbox="653 316 1353 474">Windows PowerShell: If a required parameter is not provided the runtime prompts the user for a parameter value. The prompt dialog box includes the HelpMessage text.</p>
Mandatory (named)	<p data-bbox="653 519 1050 553">Type: bool; Default value: \$false</p> <p data-bbox="653 591 1380 748">This argument specifies whether the parameter is required within the given parameter set (see ParameterSetName argument below). A value of \$true indicates that it is. A value of \$false indicates that it isn't.</p> <pre data-bbox="653 789 1157 863">param ([Parameter(Mandatory = \$true)] [string[]] \$ComputerName)</pre> <p data-bbox="653 901 1353 1058">Windows PowerShell: If a required parameter is not provided the runtime prompts the user for a parameter value. The prompt dialog box includes the HelpMessage text, if any.</p>
ParameterSetName (named)	<p data-bbox="653 1108 1241 1141">Type: string; Default value: "__AllParameterSets"</p> <p data-bbox="653 1180 1396 1382">It is possible to write a single function or cmdlet that can perform different actions for different scenarios. It does this by exposing different groups of parameters depending on the action it wants to take. Such parameter groupings are called <i>parameter sets</i>.</p> <p data-bbox="653 1420 1400 1578">The argument ParameterSetName specifies the parameter set to which a parameter belongs. This behavior means that each parameter set must have one unique parameter that is not a member of any other parameter set.</p> <p data-bbox="653 1618 1391 1775">For parameters that belong to multiple parameter sets, add a Parameter attribute for each parameter set. This allows the parameter to be defined differently for each parameter set.</p> <p data-bbox="653 1814 1359 1971">A parameter set that contains multiple positional parameters must define unique positions for each parameter. No two positional parameters can specify the same position.</p> <p data-bbox="653 2009 1294 2083">If no parameter set is specified for a parameter, the parameter belongs to all parameter sets.</p>

Parameter	Purpose
	<p>When multiple parameter sets are defined, the named argument <code>DefaultParameterSetName</code> of the attribute <code>CmdletBinding ([§12.3.5][§12.3.5])</code> is used to specify the default parameter set. The runtime uses the default parameter set if it cannot determine the parameter set to use based on the information provided by the command, or raises an exception if no default parameter set has been specified.</p> <p>The following example shows a function <code>Test</code> with a parameter declaration of two parameters that belong to two different parameter sets, and a third parameter that belongs to both sets:</p> <pre>param ([Parameter(Mandatory = \$true, ParameterSetName = "Computer")] [string[]] \$ComputerName, [Parameter(Mandatory = \$true, ParameterSetName = "User")] [string[]] \$UserName, [Parameter(Mandatory = \$true, ParameterSetName = "Computer")] [Parameter(ParameterSetName = "User")] [int] \$SharedParam = 5) if (\$PSCmdlet.ParameterSetName -eq "Computer") { # handle "Computer" parameter set } elseif (\$PSCmdlet.ParameterSetName -eq "User") { # handle "User" parameter set } ... }</pre> <p><code>Test -ComputerName "Mars","Venus" -SharedParam 10</code> <code>Test -UserName "Mary","Jack"</code> <code>Test -UserName "Mary","Jack" -SharedParam 20</code></p>
Position (named)	<p>Type: int</p> <p>This argument specifies the position of the parameter in the argument list. If this argument is not specified, the parameter name or its alias must be specified explicitly when the parameter is set. If none of the parameters of a</p>

Parameter	Purpose
	<p>function has positions, positions are assigned to each parameter based on the order in which they are received.</p> <p>The following example shows the declaration of a parameter whose value must be specified as the first argument when the function is called.</p> <pre data-bbox="653 429 1066 508">param ([Parameter(Position = 0)] [string[]] \$ComputerName)</pre>
ValueFromPipeline (named)	<p>Type: bool; Default value: \$false</p> <p>This argument specifies whether the parameter accepts input from a pipeline object. A value of \$true indicates that it does. A value of \$false indicates that it does not.</p> <p>Specify \$true if the function or cmdlet accesses the complete object, not just a property of the object.</p> <p>Only one parameter in a parameter set can declare ValueFromPipeline as \$true.</p> <p>The following example shows the parameter declaration of a mandatory parameter, \$ComputerName, that accepts the input object that is passed to the function from the pipeline.</p> <pre data-bbox="653 1204 1146 1328">param ([Parameter(Mandatory = \$true, ValueFromPipeline=\$true)] [string[]] \$ComputerName)</pre> <p>For an example of using this parameter in conjunction with the Alias attribute see [§12.3.1][§12.3.1].</p>
ValueFromPipelineByPropertyName (named)	<p>Type: bool; Default value: \$false</p> <p>This argument specifies whether the parameter takes its value from a property of a pipeline object that has either the same name or the same alias as this parameter. A value of \$true indicates that it does. A value of \$false indicates that it does not.</p> <p>Specify \$true if the following conditions are true: the parameter accesses a property of the piped object, and the property has the same name as the parameter, or the property has the same alias as the parameter.</p> <p>A parameter having ValueFromPipelineByPropertyName set to \$true need not have a parameter in the same set with ValueFromPipeline set to \$true.</p>

Parameter	Purpose
	<p>If a function has a parameter \$ComputerName, and the piped object has a ComputerName property, the value of the ComputerName property is assigned to the \$ComputerName parameter of the Function:</p> <pre data-bbox="653 361 1214 474">param ([Parameter(Mandatory = \$true, ValueFromPipelineByPropertyName = \$true)] [string[]] \$ComputerName)</pre> <p>Multiple parameters in a parameter set can define the ValueFromPipelineByPropertyName as \$true. Although, a single input object cannot be bound to multiple parameters, different properties in that input object may be bound to different parameters.</p> <p>When binding a parameter with a property of an input object, the runtime environment first looks for a property with the same name as the parameter. If such a property does not exist, the runtime environment looks for aliases to that parameter, in their declaration order, picking the first such alias for which a property exists.</p> <pre data-bbox="653 1036 1341 1643">function Process-Date { param([Parameter(ValueFromPipelineByPropertyName=\$true)] [int]\$Year, [Parameter(ValueFromPipelineByPropertyName=\$true)] [int]\$Month, [Parameter(ValueFromPipelineByPropertyName=\$true)] [int]\$Day) process { ... } }</pre> <p>Get-Date Process-Date</p>
ValueFromRemainingArguments (named)	<p>Type: bool; Default value: \$false</p> <p>This argument specifies whether the parameter accepts all of the remaining arguments that are not bound to the parameters of the function. A value of \$true indicates that it does. A value of \$false indicates that it does not.</p> <p>The following example shows a parameter \$Others that accepts all the remaining arguments of the input object that is passed to the function Test:</p>

Parameter	Purpose
	<pre>param ([Parameter(Mandatory = \$true)][int] \$p1, [Parameter(Mandatory = \$true)][int] \$p2, [Parameter(ValueFromRemainingArguments = \$true)] [string[]] \$Others)</pre> <p>Test 10 20 # \$Others has Length 0 Test 10 20 30 40 # \$Others has Length 2, value 30,40</p>

An implementation may define other attributes as well.

The following attributes are provided as well:

- **HelpMessageBaseName**: Specifies the location where resource identifiers reside. For example, this parameter could specify a resource assembly that contains Help messages that are to be localized.
- **HelpMessageResourceId**: Specifies the resource identifier for a Help message.

12.3.8 The PSDefaultValue attribute

This attribute is used in a *script-parameter* to provide additional information about the parameter. The attribute is used in an implementation defined manner. The following arguments are used to define the characteristics of the parameter:

[Expand table](#)

Parameter	Purpose
Name	
Help (named)	<p>Type: string</p> <p>This argument specifies a message that is intended to contain a short description of the default value of a parameter. This message is used in an implementation-defined manner.</p> <p>Windows PowerShell: The message is used as part of the description of the parameter for the help topic displayed by the [Get-Help](xref:Microsoft.PowerShell.Core.Get-Help) cmdlet.</p>
Value (named)	<p>Type: object</p> <p>This argument specifies a value that is intended to be the default value of a parameter. The value is used in an implementation-defined manner.</p> <p>Windows PowerShell: The value is used as part of the description of the parameter for the help topic displayed by the [Get-Help](xref:Microsoft.PowerShell.Core.Get-Help) cmdlet when the Help property is not specified.</p>

12.3.9 The SupportsWildcards attribute

This attribute is used in a *script-parameter* to provide additional information about the parameter. The attribute is used in an implementation defined manner.

This attribute is used as part of the description of the parameter for the help topic displayed by the [Get-Help](#) cmdlet.

12.3.10 The ValidateCount attribute

This attribute is used in a *script-parameter* to specify the minimum and maximum number of argument values that the parameter can accept. The following arguments are used to define the characteristics of the parameter:

[] [Expand table](#)

Parameter Name	Purpose
MinLength (position 0)	Type: int This argument specifies the minimum number of argument values allowed.
MaxLength (position 1)	Type: int This argument specifies the maximum number of argument values allowed.

In the absence of this attribute, the parameter's corresponding argument value list can be of any length.

Consider a function call `Test` that has the following param block, and which is called as shown:

```
PowerShell

param (
    [ValidateCount(2, 5)]
    [int[]] $Values
)

Temp 10, 20, 30
Temp 10          # too few argument values
Temp 10, 20, 30, 40, 50, 60      # too many argument values

[ValidateCount(3, 4)]$Array = 1..3
```

```
$Array = 10                      # too few argument values  
$Array = 1..100                    # too many argument values
```

12.3.11 The ValidateLength attribute

This attribute is used in a *script-parameter* or *variable* to specify the minimum and maximum length of the parameter's argument, which must have type string. The following arguments are used to define the characteristics of the parameter:

[\[+\] Expand table](#)

Parameter Name	Purpose
MinLength (position 0)	Type: int This argument specifies the minimum number of characters allowed.
MaxLength (position 1)	Type: int This argument specifies the maximum number of characters allowed.

In the absence of this attribute, the parameter's corresponding argument can be of any length.

Consider a function call `Test` that has the following param block, and which is called as shown:

```
PowerShell  
  
param ( [Parameter(Mandatory = $true)]  
[ValidateLength(3,6)]  
[string[]] $ComputerName )  
  
Test "Thor", "Mars"      # length is ok  
Test "Io", "Mars"        # "Io" is too short  
Test "Thor", "Jupiter"   # "Jupiter" is too long
```

12.3.12 The ValidateNotNull attribute

This attribute is used in a *script-parameter* or *variable* to specify that the argument of the parameter cannot be `$null` or be a collection containing a null-valued element.

Consider a function call `Test` that has the following param block, and which is called as shown:

PowerShell

```
param (
    [ValidateNotNull()]  
    [string[]] $Names  
)  
  
Test "Jack", "Jill"      # ok  
Test "Jane", $null        # $null array element value not allowed  
Test $null                # null array not allowed  
  
[ValidateNotNull()]$Name = "Jack" # ok  
$Name = $null              # null value not allowed
```

12.3.13 The ValidateNotNullOrEmpty attribute

This attribute is used in a *script-parameter* or *variable* to specify that the argument if the parameter cannot be \$null, an empty string, or an empty array, or be a collection containing a \$null-valued or empty string element.

Consider a function call `Test` that has the following param block, and which is called as shown:

PowerShell

```
param (  
    [ValidateNotNullOrEmpty()]  
    [string[]] $Names  
)  
  
Test "Jack", "Jill"      # ok  
Test "Mary", ""           # empty string not allowed  
Test "Jane", $null        # $null array element value not allowed  
Test $null                # null array not allowed  
Test @()                  # empty array not allowed  
  
[ValidateNotNullOrEmpty()]$Name = "Jack" # ok  
$Name = ""                 # empty string not allowed  
$Name = $null              # null value not allowed
```

12.3.14 The ValidatePattern attribute

This attribute is used in a *script-parameter* or *variable* to specify a regular expression for matching the pattern of the parameter's argument. The following arguments are used to define the characteristics of the parameter:

Parameter Name	Purpose
RegexString (position 0)	Type: String A regular expression that is used to validate the parameter's argument
Options (named)	Type: Regular-Expression-Option See [§4.2.6.4][§4.2.6.4] for the allowed values.

If the argument is a collection, each element in the collection must match the pattern.

Consider a function call `Test` that has the following param block, and which is called as shown:

```
PowerShell

param (
    [ValidatePattern('^\[A-Z\]\[1-5\]\[0-9\]$')]
    [string] $Code,

    [ValidatePattern('^\(0x|0X\)(\[A-F\]|\[a-f\]|\[0-9\])(\[A-F\]|\[a-f\]|\[0-9\])$')]
    [string] $HexNum,

    [ValidatePattern('^\[+|-?\[1-9\]$')]
    [int] $Minimum
)

Test -C A12 # matches pattern
Test -C A63 # does not match pattern

Test -H 0x4f # matches pattern
Test -H "0XB2" # matches pattern
Test -H 0xK3 # does not match pattern

Test -M -4 # matches pattern
Test -M "+7" # matches pattern
Test -M -12 # matches pattern, but is too long

[ValidatePattern('^\[a-z\]\[a-z0-9\]\*$')]$ident = "abc"
$ident = "123" # does not match pattern
```

12.3.15 The ValidateRange attribute

This attribute is used in a *script-parameter* or *variable* to specify the minimum and maximum values of the parameter's argument. The following arguments are used to define the characteristics of the parameter:

Parameter Name	Purpose
MinRange (position 0)	Type: object This argument specifies the minimum value allowed.
MaxRange (position 1)	Type: object This argument specifies the maximum value allowed.

In the absence of this attribute, there is no range restriction.

Consider a function call `Test1` that has the following param block, and which is called as shown:

PowerShell

```
param (
    [Parameter(Mandatory = $true)]
    [ValidateRange(1, 10)]
    [int] $StartValue
)

Test1 2
Test1 -St 7
Test1 -3 # value is too small
Test1 12 # value is too large
```

Consider a function call `Test2` that has the following param block and calls:

PowerShell

```
param (
    [Parameter(Mandatory = $true)]
    [ValidateRange("b", "f")]
    [string] $Name
)

Test2 "Bravo" # ok
Test2 "Alpha" # value compares less than the minimum
Test2 "Hotel" # value compares greater than the maximum
```

Consider a function call `Test3` that has the following param block, and which is called as shown:

PowerShell

```

param (
    [Parameter(Mandatory = $true)]
    [ValidateRange(0.002, 0.003)]
    [double] $Distance
)

Test3 0.002
Test3 0.0019    # value is too small
Test3 "0.005"   # value is too large

[ValidateRange(13, 19)]$teenager = 15
$teenager = 20  # value is too large

```

12.3.16 The ValidateScript attribute

This attribute is used in a *script-parameter* or *variable* to specify a script that is to be used to validate the parameter's argument.

The argument in position 1 is a *script-block-expression*.

Consider a function call `Test` that has the following param block, and which is called as shown:

PowerShell

```

param (
    [Parameter(Mandatory = $true)]
    [ValidateScript( { ($_. -ge 1 -and $_ -le 3) -or ($_. -ge 20) })]
    [int] $Count
)

Test 2 # ok, valid value
Test 25 # ok, valid value
Test 5 # invalid value
Test 0 # invalid value

[ValidateScript({$.Length --gt 7})]$password = "password" # ok
$password = "abc123" # invalid value

```

12.3.17 The ValidateSet attribute

This attribute is used in a *script-parameter* or *variable* to specify a set of valid values for the argument of the parameter. The following arguments are used to define the characteristics of the parameter:

Parameter Name	Purpose
ValidValues (position 0)	Type: string[] The set of valid values.
IgnoreCase (named)	Type: bool; Default value: \$true Specifies whether case should be ignored for parameters of type string.

If the parameter has an array type, every element of the corresponding argument array must match an element of the value set.

Consider a function call `Test` that has the following param block, and which is called as shown:

PowerShell

```
param ( [ValidateSet("Red", "Green", "Blue")]
    [string] $Color,
    [ValidateSet("up", "down", "left", "right", IgnoreCase =
        $false)]
    [string] $Direction
)

Test -Col "RED"      # case is ignored, is a member of the set
Test -Col "white"   # case is ignored, is not a member of the set

Test -Dir "up"       # case is not ignored, is a member of the set
Test -Dir "Up"       # case is not ignored, is not a member of the set

[ValidateSet(("Red", "Green", "Blue"))]$color = "RED" # ok, case is ignored
$color = "Purple"  # case is ignored, is not a member of the set
```

13. Cmdlets

Article • 01/08/2025

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

A cmdlet is a single-feature command that manipulates objects in PowerShell. Cmdlets can be recognized by their name format, a verb and noun separated by a dash (-), such as `Get-Help`, `Get-Process`, and `Start-Service`. A *verb pattern* is a verb expressed using wildcards, as in `w*`. A *noun pattern* is a noun expressed using wildcards, as in `event`.

Cmdlets should be simple and be designed to be used in combination with other cmdlets. For example, **Get** cmdlets should only retrieve data, **Set** cmdlets should only establish or change data, **Format** cmdlets should only format data, and **Out** cmdlets should only direct the output to a specified destination.

For each cmdlet, provide a help file that can be accessed by typing:

```
Get-Help *cmdlet-name* -Detailed
```

The detailed view of the cmdlet help file should include a description of the cmdlet, the command syntax, descriptions of the parameters, and an example that demonstrate the use of that cmdlet.

Cmdlets are used similarly to operating system commands and utilities. PowerShell commands are not case-sensitive.

Note

Editor's note: The original document contains a list of cmdlet with descriptions, syntax diagrams, parameter definitions, and examples. This information is incomplete and out dated. For current information about cmdlet, consult the [Reference section of the PowerShell documentation](#).

13.1 Common parameters

The *common parameters* are a set of cmdlet parameters that can be used with any cmdlet. They are implemented by the PowerShell runtime environment itself, not by the cmdlet developer, and they are automatically available to any cmdlet or function that uses the [Parameter attribute](#) ([§12.3.7](#)) or [CmdletBinding attribute](#) ([§12.3.5](#)).

Although the common parameters are accepted by any cmdlet, they might not have any semantics for that cmdlet. For example, if a cmdlet does not generate any verbose output, using the **Verbose** common parameter has no effect.

Several common parameters override system defaults or preferences that can be set via preference variables ([§2.3.2.3](#)). Unlike the preference variables, the common parameters affect only the commands in which they are used.

Note

Editor's note: The original document contains a list of the Common Parameters. This information is incomplete and out dated. For current information see [about_CommonParameters](#).

A. Comment-Based Help

Article • 01/08/2025

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

PowerShell provides a mechanism for programmers to document their scripts using special comment directives. Comments using such syntax are called *help comments*. The cmdlet [Get-Help](#) generates documentation from these directives.

A.1 Introduction

A help comment contains a *help directive* of the form `.name` followed on one or more subsequent lines by the help content text. The help comment can be made up of a series of *single-line-comments* or a *delimited-comment* ([§2.2.3](#)). The set of comments comprising the documentation for a single entity is called a *help topic*.

For example,

PowerShell

```
# <help-directive-1>
# <help-content-1>
...

```

```
# <help-directive-n>
# <help-content-n>
```

or

PowerShell

```
<#
<help-directive-1>
<help-content-1>
...
<help-directive-n>
<help-content-n>
#>
```

All of the lines in a help topic must be contiguous. If a help topic follows a comment that is not part of that topic, there must be at least one blank line between the two.

The directives can appear in any order, and some of the directives may appear multiple times.

Directive names are not case-sensitive.

When documenting a function, help topics may appear in one of three locations:

- Immediately before the function definition with no more than one blank line between the last line of the function help and the line containing the function statement.
- Inside the function's body immediately following the opening curly bracket.
- Inside the function's body immediately preceding the closing curly bracket.

When documenting a script file, help topics may appear in one of two locations:

- At the beginning of the script file, optionally preceded by comments and blank lines only. If the first item in the script after the help is a function definition, there must be at least two blank lines between the end of the script help and that function declaration. Otherwise, the help will be interpreted as applying to the function instead of the script file.
- At the end of the script file.

A.2 Help directives

A.2.1 .DESCRIPTION

Syntax:

```
Syntax  
.DESCRIPTION
```

Description:

This directive allows for a detailed description of the function or script. (The `.SYNOPSIS` directive ([§A.2.11](#)) is intended for a brief description.) This directive can be used only once in each topic.

Examples:

```
PowerShell  
<#  
.DESCRIPTION  
Computes Base to the power Exponent. Supports non-negative integer  
powers only.  
#>
```

A.2.2 .EXAMPLE

Syntax:

```
Syntax  
.EXAMPLE
```

Description:

This directive allows an example of command usage to be shown.

If this directive occurs multiple times, each associated help content block is displayed as a separate example.

Examples:

```
PowerShell  
<#  
.EXAMPLE  
Get-Power 3 4  
81
```

```
.EXAMPLE  
Get-Power -Base 3 -Exponent 4  
81  
#>
```

A.2.3 .EXTERNALHELP

Syntax:

```
Syntax  
.EXTERNALHELP <XMLHelpFilePath>
```

Description:

This directive specifies the path to an XML-based help file for the script or function.

Although comment-based help is easier to implement, XML-based Help is required if more precise control is needed over help content or if help topics are to be translated into multiple languages. The details of XML-based help are not defined by this specification.

Examples:

```
PowerShell  
<#  
.EXTERNALHELP C:\MyScripts\Update-Month-Help.xml  
#>
```

A.2.4 .FORWARDHELPCATEGORY

Syntax:

```
Syntax  
.FORWARDHELPCATEGORY <Category>
```

Description:

Specifies the help category of the item in **ForwardHelpTargetName** ([§A.2.5](#)). Valid values are **Alias**, **All**, **Cmdlet**, **ExternalScript**, **FAQ**, **Filter**, **Function**, **General**, **Glossary**, **HelpFile**, **Provider**, and **ScriptCommand**. Use this directive to avoid conflicts when there are commands with the same name.

Examples:

See §A.2.5.

A.2.5 .FORWARDHELPTARGETNAME

Syntax:

Syntax

```
.FORWARDHELPTARGETNAME <Command-Name>
```

Description:

Redirects to the help topic specified by `<Command-Name>`.

Examples:

PowerShell

```
function Help {
    #
    .FORWARDHELPTARGETNAME Get-Help
    .FORWARDHELPCATEGORY Cmdlet
    #
    ...
}
```

The command `Get-Help help` is treated as if it were `Get-Help Get-Help` instead.

A.2.6 .INPUTS

Syntax:

Syntax

```
.INPUTS
```

Description:

The pipeline can be used to pipe one or more objects to a script or function. This directive is used to describe such objects and their types.

If this directive occurs multiple times, each associated help content block is collected in the one documentation entry, in the directives' lexical order.

Examples:

PowerShell

```
<#
.INPUTS
None. You cannot pipe objects to Get-Power.

.INPUTS
For the Value parameter, one or more objects of any kind can be written
to the pipeline. However, the object is converted to a string before it
is added to the item.
#>
function Process-Thing {
    param (
        [Parameter(ValueFromPipeline=$true)]
        [Object[]]$Value,
        ...
    )
    ...
}
```

A.2.7 .LINK

Syntax:

Syntax

```
.LINK
```

Description:

This directive specifies the name of a related topic.

If this directive occurs multiple times, each associated help content block is collected in the one documentation entry, in the directives' lexical order.

The Link directive content can also include a URI to an online version of the same help topic. The online version is opens when Get-Help is invoked with the Online parameter. The URI must begin with "http" or "https".

Examples:

PowerShell

```
<#
.INPUTS
None. You cannot pipe objects to Get-Power.

.INPUTS
For the Value parameter, one or more objects of any kind can be written
to the pipeline. However, the object is converted to a string before it
is added to the item.
#>
function Process-Thing {
    param (
        [Parameter(ValueFromPipeline=$true)]
        [Object[]]$Value,
        ...
    )
    ...
}

#>
#>
#>
```

```
.LINK  
Set-ProcedureName  
#>
```

A.2.8 .NOTES

Syntax:

```
Syntax
```

```
.NOTES
```

Description:

This directive allows additional information about the function or script to be provided.
This directive can be used only once in each topic.

Examples:

```
PowerShell
```

```
<#  
.NOTES  
*arbitrary text goes here*  
#>
```

A.2.9 .OUTPUTS

Syntax:

```
Syntax
```

```
.OUTPUTS
```

Description:

This directive is used to describe the objects output by a command.

If this directive occurs multiple times, each associated help content block is collected in the one documentation entry, in the directives' lexical order.

Examples:

PowerShell

```
<#
.OBJECTS
double - Get-Power returns Base to the power Exponent.

.OBJECTS
None unless the -PassThru switch parameter is used.
#>
```

A.2.10 .PARAMETER

Syntax:

Syntax

```
.PARAMETER <Parameter-Name>
```

Description:

This directive allows for a detailed description of the given parameter. This directive can be used once for each parameter. Parameter directives can appear in any order in the comment block; however, the order in which their corresponding parameters are actually defined in the source determines the order in which the parameters and their descriptions appear in the resulting documentation.

An alternate format involves placing a parameter description comment immediately before the declaration of the corresponding parameter variable's name. If the source contains both a parameter description comment and a Parameter directive, the description associated with the Parameter directive is used.

Examples:

PowerShell

```
<#
.PARAMETER Base
The integer value to be raised to the Exponent-th power.

.PARAMETER Exponent
The integer exponent to which Base is to be raised.
#>

function Get-Power {
    param ([long]$Base, [int]$Exponent)
    ...
}
```

```
function Get-Power {
    param ([long]
        # The integer value to be raised to the Exponent-th power.
        $Base,
        [int]
        # The integer exponent to which Base is to be raised.
        $Exponent
    )
    ...
}
```

A.2.11 .SYNOPSIS

Syntax:

```
PowerShell
```

```
.SYNOPSIS
```

Description:

This directive allows for a brief description of the function or script. (The `.DESCRIPTION` directive ([§A.2.1](#)) is intended for a detailed description.) This directive can be used only once in each topic.

Examples:

```
PowerShell
```

```
<#
.SYNOPSIS
Computes Base to the power Exponent.
#>
```

B. Grammar

Article • 01/08/2025

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

This appendix contains summaries of the lexical and syntactic grammars found in the main document.

Tip

The `~opt~` notation in the syntax definitions indicates that the lexical entity is optional in the syntax.

B.1 Lexical grammar

Syntax

```
input:  
    input-elements~opt~ signature-block~opt~  
  
input-elements:  
    input-element  
    input-elements input-element  
  
input-element:
```

```
whitespace
comment
token

signature-block:
    signature-begin signature signature-end

signature-begin:
    new-line-character # SIG # Begin signature block new-line-character

signature:
    base64 encoded signature blob in multiple single-line-comments

signature-end:
    new-line-character # SIG # End signature block new-line-character
```

B.1.1 Line terminators

Syntax

```
new-line-character:
    Carriage return character (U+000D)
    Line feed character (U+000A)
    Carriage return character (U+000D) followed by line feed character
(U+000A)

new-lines:
    new-line-character
    new-lines new-line-character
```

B.1.2 Comments

Syntax

```
comment:
    single-line-comment
    requires-comment
    delimited-comment

single-line-comment:
    # input-characters~opt~

input-characters:
    input-character
    input-characters input-character

input-character:
    Any Unicode character except a new-line-character
```

```

requires-comment:
    #Requires whitespace command-arguments

dash:
    - (U+002D)
    EnDash character (U+2013)
    EmDash character (U+2014)
    Horizontal bar character (U+2015)

dashdash:
    dash dash

delimited-comment:
    <# delimited-comment-text~opt~ hashes >

delimited-comment-text:
    delimited-comment-section
    delimited-comment-text delimited-comment-section

delimited-comment-section:
    >
    hashes~opt~ not-greater-than-or-hash

hashes:
    #
    hashes #

not-greater-than-or-hash:
    Any Unicode character except > or #

```

B.1.3 White space

Syntax

```

whitespace:
    Any character with Unicode class Zs, Zl, or Zp
    Horizontal tab character (U+0009)
    Vertical tab character (U+000B)
    Form feed character (U+000C)
    ` (The backtick character U+0060) followed by new-line-character

```

B.1.4 Tokens

Syntax

```

token:
    keyword
    variable
    command

```

```
command-parameter
command-argument-token
integer-literal
real-literal
string-literal
type-literal
operator-or-punctuator
```

B.1.5 Keywords

Syntax

```
keyword: one of
begin      break      catch      class
continue    data       define     do
dynamicparam else      elseif     end
exit        filter     finally   for
foreach     from      function  if
in          inlinescript parallel param
process    return     switch    throw
trap       try       until     using
var        while     workflow
```

B.1.6 Variables

Syntax

```
variable:
$$
$?
$^
$  variable-scope~opt~  variable-characters
@  variable-scope~opt~  variable-characters
braced-variable

braced-variable:
${  variable-scope~opt~  braced-variable-characters  }

variable-scope:
Global:
Local:
Private:
Script:
Using:
Workflow:
variable-namespace

variable-namespace:
```

```

variable-characters  :

variable-characters:
    variable-character
    variable-characters  variable-character

variable-character:
    A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nd
    _   (The underscore character U+005F)
    ?

braced-variable-characters:
    braced-variable-character
    braced-variable-characters  braced-variable-character

braced-variable-character:
    Any Unicode character except
        }   (The closing curly brace character U+007D)
        `   (The backtick character U+0060)
    escaped-character

escaped-character:
    `   (The backtick character U+0060) followed by any Unicode character

```

B.1.7 Commands

Syntax

```

generic-token:
    generic-token-parts

generic-token-parts:
    generic-token-part
    generic-token-parts generic-token-part

generic-token-part:
    expandable-string-literal
    verbatim-here-string-literal
    variable
    generic-token-char

generic-token-char:
    Any Unicode character except
        {   }   (   )   ;   ,   |   &   $
        `   (The backtick character U+0060)
    double-quote-character
    single-quote-character
    whitespace
    new-line-character
    escaped-character

```

```
generic-token-with-subexpr-start:  
    generic-token-parts $()
```

B.1.8 Parameters

Syntax

```
command-parameter:  
    dash first-parameter-char parameter-chars colon~opt~  
  
first-parameter-char:  
    A Unicode character of classes Lu, Ll, Lt, Lm, or Lo  
    _ (The underscore character U+005F)  
    ?  
  
parameter-chars:  
    parameter-char  
    parameter-chars parameter-char  
  
parameter-char:  
    Any Unicode character except  
        { } ( ) ; , | & . [  
    colon  
    whitespace  
    new-line-character  
  
colon:  
    : (The colon character U+003A)  
  
verbatim-command-argument-chars:  
    verbatim-command-argument-part  
    verbatim-command-argument-chars verbatim-command-argument-part  
  
verbatim-command-argument-part:  
    verbatim-command-string  
    & non-ampersand-character  
    Any Unicode character except  
        |  
    new-line-character  
  
non-ampersand-character:  
    Any Unicode character except &  
  
verbatim-command-string:  
    double-quote-character non-double-quote-chars  
    double-quote-character  
  
non-double-quote-chars:  
    non-double-quote-char  
    non-double-quote-chars non-double-quote-char  
  
non-double-quote-char:
```

Any Unicode character except
double-quote-character

B.1.9 Literals

Syntax

```
literal:  
    integer-literal  
    real-literal  
    string-literal
```

B.1.9.1 Integer Literals

Syntax

```
integer-literal:  
    decimal-integer-literal  
    hexadecimal-integer-literal

decimal-integer-literal:  
    decimal-digits numeric-type-suffix~opt~ numeric-multiplier~opt~

decimal-digits:  
    decimal-digit  
    decimal-digit decimal-digits

decimal-digit: one of  
    0 1 2 3 4 5 6 7 8 9

numeric-type-suffix:  
    long-type-suffix  
    decimal-type-suffix

hexadecimal-integer-literal:  
    0x hexadecimal-digits long-type-suffix~opt~  
    numeric-multiplier~opt~

hexadecimal-digits:  
    hexadecimal-digit  
    hexadecimal-digit decimal-digits

hexadecimal-digit: one of  
    0 1 2 3 4 5 6 7 8 9 a b c d e f

long-type-suffix:  
    l
```

```
numeric-multiplier: one of
  kb mb gb tb pb
```

B.1.9.2 Real Literals

Syntax

```
real-literal:
  decimal-digits . decimal-digits exponent-part~opt~ decimal-type-
suffix~opt~ numeric-multiplier~opt~
  . decimal-digits exponent-part~opt~ decimal-type-suffix~opt~ numeric-
multiplier~opt~
  decimal-digits exponent-part decimal-type-suffix~opt~ numeric-
multiplier~opt~

exponent-part:
  e sign~opt~ decimal-digits

sign: one of
  +
  dash

decimal-type-suffix:
  d
  l
```

B.1.9.3 String Literals

Syntax

```
string-literal:
  expandable-string-literal
  expandable-here-string-literal
  verbatim-string-literal
  verbatim-here-string-literal

expandable-string-literal:
  double-quote-character expandable-string-characters~opt~ dollars~opt~
double-quote-character

double-quote-character:
  " (U+0022)
  Left double quotation mark (U+201C)
  Right double quotation mark (U+201D)
  Double low-9 quotation mark (U+201E)

expandable-string-characters:
  expandable-string-part
  expandable-string-characters
```

```
expandable-string-part

expandable-string-part:
    Any Unicode character except
        $
        double-quote-character
        ` (The backtick character U+0060)
    braced-variable
    $ Any Unicode character except
        (
        {
        double-quote-character
        ` (The backtick character U+0060)*
    $ escaped-character
    escaped-character
    double-quote-character double-quote-character

dollars:
    $
    dollars $

expandable-here-string-literal:
    @ double-quote-character whitespace~opt~ new-line-character
        expandable-here-string-characters~opt~ new-line-character double-
        quote-character @

expandable-here-string-characters:
    expandable-here-string-part
    expandable-here-string-characters expandable-here-string-part

expandable-here-string-part:
    Any Unicode character except
        $
        new-line-character
    braced-variable
    $ Any Unicode character except
        (
        new-line-character
    $ new-line-character Any Unicode character except double-quote-char
    $ new-line-character double-quote-char Any Unicode character except @
    new-line-character Any Unicode character except double-quote-char
    new-line-character double-quote-char Any Unicode character except @

expandable-string-with-subexpr-start:
    double-quote-character expandable-string-chars~opt~ $(

expandable-string-with-subexpr-end:
    double-quote-char

expandable-here-string-with-subexpr-start:
    @ double-quote-character whitespace~opt~ new-line-character expandable-
    here-string-chars~opt~ $(

expandable-here-string-with-subexpr-end:
    new-line-character double-quote-character @
```

```

verbatim-string-literal:
    single-quote-character verbatim-string-characters~opt~ single-quote-char

single-quote-character:
    ' (U+0027)
    Left single quotation mark (U+2018)
    Right single quotation mark (U+2019)
    Single low-9 quotation mark (U+201A)
    Single high-reversed-9 quotation mark (U+201B)

verbatim-string-characters:
    verbatim-string-part
    verbatim-string-characters verbatim-string-part

verbatim-string-part:
    *Any Unicode character except* single-quote-character
    single-quote-character single-quote-character

verbatim-here-string-literal:
    @ single-quote-character whitespace~opt~ new-line-character
        verbatim-here-string-characters~opt~ new-line-character
            single-quote-character *@*

verbatim-*here-string-characters:
    verbatim-here-string-part
    verbatim-here-string-characters verbatim-here-string-part

verbatim-here-string-part:
    Any Unicode character except* new-line-character
    new-line-character Any Unicode character except single-quote-character
    new-line-character single-quote-character Any Unicode character except
    @

```

B.1.10 Simple Names

Syntax

```

simple-name:
    simple-name-first-char simple-name-chars

simple-name-first-char:
    A Unicode character of classes Lu, Ll, Lt, Lm, or Lo
    _ (The underscore character U+005F)

simple-name-chars:
    simple-name-char
    simple-name-chars simple-name-char

simple-name-char:

```

A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nd
_ (The underscore character U+005F)

B.1.11 Type Names

Syntax

```
type-name:  
    type-identifier  
    type-name . type-identifier  
  
type-identifier:  
    type-characters  
  
type-characters:  
    type-character  
    type-characters type-character  
  
type-character:  
    A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nd  
    _ (The underscore character U+005F)  
  
array-type-name:  
    type-name [  
  
generic-type-name:  
    type-name [
```

B.1.12 Operators and punctuators

Syntax

```
operator-or-punctuator: one of  
    { } [ ] ( ) @  
    && || & | , ++ .. :: .  
    ! * / % +  
    dash dashdash  
    dash and dash band dash bnot dash bor  
    dash bxor dash not dash or dash xor  
    assignment-operator  
    merging-redirection-operator  
    file-redirection-operator  
    comparison-operator  
    format-operator  
  
assignment-operator: one of  
    = dash = += *= /= %=  
  
file-redirection-operator: one of
```

```

>  >>  2>  2>>  3>  3>>  4>  4>>
5>  5>>  6>  6>>  *>  *>>  <

merging-redirection-operator: one of
  *>&1  2>&1  3>&1  4>&1  5>&1  6>&1
  *>&2  1>&2  3>&2  4>&2  5>&2  6>&2

comparison-operator: one of
  dash as           dash ccontains      dash ceq
  dash cge          dash cgt          dash cle
  dash clike         dash clt          dash cmatch
  dash cne          dash cnotcontains  dash cnotlike
  dash cnotmatch    dash contains      dash creplace
  dash csplit        dash eq           dash ge
  dash gt            dash icontains   dash ieq
  dash ige           dash igt          dash ile
  dash ilike         dash ilt          dash imatch
  dash in             dash ine          dash inotcontains
  dash inotlike      dash inotmatch   dash ireplace
  dash is             dash isnot         dash isplit
  dash join          dash le           dash like
  dash lt             dash match        dash ne
  dash notcontains   dash notin        dash notlike
  dash notmatch      dash replace      dash shl*
  dash shr            dash split

format-operator:
  dash f

```

B.2 Syntactic grammar

B.2.1 Basic concepts

Syntax	
script-file:	script-block
module-file:	script-block
interactive-input:	script-block
data-file:	statement-list

B.2.2 Statements

Syntax

```
script-block:
    param-block~opt~ statement-terminators~opt~ script-block-body~opt~

param-block:
    new-lines~opt~ attribute-list~opt~ new-lines~opt~ param new-lines~opt~
        ( parameter-list~opt~ new-lines~opt~ )

parameter-list:
    script-parameter
    parameter-list new-lines~opt~ , script-parameter

script-parameter:
    new-lines~opt~ attribute-list~opt~ new-lines~opt~ variable script-
parameter-default~opt~

script-parameter-default:
    new-lines~opt~ = new-lines~opt~ expression

script-block-body:
    named-block-list
    statement-list

named-block-list:
    named-block
    named-block-list named-block

named-block:
    block-name statement-block statement-terminators~opt~

block-name: one of
    dynamicparam begin process end

statement-block:
    new-lines~opt~ { statement-list~opt~ new-lines~opt~ }

statement-list:
    statement
    statement-list statement

statement:
    if-statement
    label~opt~ labeled-statement
    function-statement
    flow-control-statement statement-terminator
    trap-statement
    try-statement
    data-statement
    inlinescript-statement
    parallel-statement
    sequence-statement
    pipeline statement-terminator
```

```
statement-terminator:
;
new-line-character

statement-terminators:
    statement-terminator
    statement-terminators statement-terminator

if-statement:
    if new-lines~opt~ ( new-lines~opt~ pipeline new-lines~opt~ ) statement-
block
        elseif-clauses~opt~ else-clause~opt~

elseif-clauses:
    elseif-clause
    elseif-clauses elseif-clause

elseif-clause:
    new-lines~opt~ elseif new-lines~opt~ ( new-lines~opt~ pipeline new-
lines~opt~ ) statement-block

else-clause:
    new-lines~opt~ else statement-block

labeled-statement:
    switch-statement
    foreach-statement
    for-statement
    while-statement
    do-statement

switch-statement:
    switch new-lines~opt~ switch-parameters~opt~ switch-condition switch-
body

switch-parameters:
    switch-parameter
    switch-parameters switch-parameter

switch-parameter:
    -Regex
    -Wildcard
    -Exact
    -CaseSensitive
    -Parallel

switch-condition:
    ( new-lines~opt~ pipeline new-lines~opt~ )
    -file new-lines~opt~ switch-filename

switch-filename:
    command-argument
    primary-expression

switch-body:
```

```
new-lines~opt~ { new-lines~opt~ switch-clauses }

switch-clauses:
    switch-clause
    switch-clauses switch-clause

switch-clause:
    switch-clause-condition statement-block statement-terminators~opt~

switch-clause-condition:
    command-argument
    primary-expression

foreach-statement:
    foreach new-lines~opt~ foreach-parameter~opt~ new-lines~opt~
        ( new-lines~opt~ variable new-lines~opt~ in new-lines~opt~ pipeline
          new-lines~opt~ ) statement-block

foreach-parameter:
    -parallel

for-statement:
    for new-lines~opt~ (
        new-lines~opt~ for-initializer~opt~ statement-terminator
        new-lines~opt~ for-condition~opt~ statement-terminator
        new-lines~opt~ for-iterator~opt~
        new-lines~opt~ ) statement-block
    for new-lines~opt~ (
        new-lines~opt~ for-initializer~opt~ statement-terminator
        new-lines~opt~ for-condition~opt~
        new-lines~opt~ ) statement-block
    for new-lines~opt~ (
        new-lines~opt~ for-initializer~opt~
        new-lines~opt~ ) statement-block

for-initializer:
    pipeline

for-condition:
    pipeline

for-iterator:
    pipeline

while-statement:
    while new-lines~opt~ ( new-lines~opt~ while-condition new-lines~opt~ )
statement-block

do-statement:
    do statement-block new-lines~opt~ while new-lines~opt~ ( while-condition
      new-lines~opt~ )
    do statement-block new-lines~opt~ until new-lines~opt~ ( while-condition
      new-lines~opt~ )

while-condition:
```

```
new-lines~opt~ pipeline

function-statement:
    function new-lines~opt~ function-name function-parameter-
declaration~opt~ { script-block }
    filter new-lines~opt~ function-name function-parameter-declaration~opt~
{ script-block }
    workflow new-lines~opt~ function-name function-parameter-
declaration~opt~ { script-block }

function-name:
    command-argument

function-parameter-declaration:
    new-lines~opt~ ( parameter-list new-lines~opt~ )

flow-control-statement:
    break label-expression~opt~
    continue label-expression~opt~
    throw pipeline~opt~
    return pipeline~opt~
    exit pipeline~opt~

label-expression:
    simple-name
    unary-expression

trap-statement:
    trap new-lines~opt~ type-literal~opt~ new-lines~opt~ statement-block

try-statement:
    try statement-block catch-clauses
    try statement-block finally-clause
    try statement-block catch-clauses finally-clause

catch-clauses:
    catch-clause
    catch-clauses catch-clause

catch-clause:
    new-lines~opt~ catch catch-type-list~opt~ statement-block

catch-type-list:
    new-lines~opt~ type-literal
    catch-type-list new-lines~opt~, new-lines~opt~ type-literal

finally-clause:
    new-lines~opt~ finally statement-block

data-statement:
    data new-lines~opt~ data-name data-commands-allowed~opt~
    statement-block

data-name:
    simple-name
```

```
data-commands-allowed:
    new-lines~opt~ -SupportedCommand data-commands-list

data-commands-list:
    new-lines~opt~ data-command
    data-commands-list , new-lines~opt~ data-command

data-command:
    command-name-expr

inlinescript-statement:
    inlinescript statement-block

parallel-statement:
    parallel statement-block

sequence-statement:
    sequence statement-block

pipeline:
    assignment-expression
    expression redirections~opt~ pipeline-tail~opt~
    command verbatim-command-argument~opt~ pipeline-tail~opt~

assignment-expression:
    expression assignment-operator statement

pipeline-tail:
    | new-lines~opt~ command
    | new-lines~opt~ command pipeline-tail

command:
    command-name command-elements~opt~
    command-invocation-operator command-module~opt~ command-name-expr
    command-elements~opt~

command-invocation-operator: one of
    & .

command-module:
    primary-expression

command-name:
    generic-token
    generic-token-with-subexpr

generic-token-with-subexpr:
    No whitespace is allowed between ) and command-name.
    generic-token-with-subexpr-start statement-list~opt~ ) command-name

command-name-expr:
    command-name
    primary-expression
```

```

command-elements:
  command-element
  command-elements command-element

command-element:
  command-parameter
  command-argument
  redirection

command-argument:
  command-name-expr

verbatim-command-argument:
  --% verbatim-command-argument-chars

redirections:
  redirection
  redirections redirection

redirection:
  merging-redirection-operator
  file-redirection-operator redirected-file-name

redirected-file-name:
  command-argument
  primary-expression

```

B.2.3 Expressions

Syntax

```

expression:
  logical-expression

logical-expression:
  bitwise-expression
  logical-expression -and new-lines~opt~ bitwise-expression
  logical-expression -or new-lines~opt~ bitwise-expression
  logical-expression -xor new-lines~opt~ bitwise-expression

bitwise-expression:
  comparison-expression
  bitwise-expression -band new-lines~opt~ comparison-expression
  bitwise-expression -bor new-lines~opt~ comparison-expression
  bitwise-expression -bxor new-lines~opt~ comparison-expression

comparison-expression:
  additive-expression
  comparison-expression comparison-operator new-lines~opt~
  additive-expression

additive-expression:

```

```
multiplicative-expression
additive-expression + new-lines~opt~ multiplicative-expression
additive-expression dash new-lines~opt~ multiplicative-expression

multiplicative-expression:
    format-expression
    multiplicative-expression \ new-lines~opt~ format-expression
    multiplicative-expression / new-lines~opt~ format-expression
    multiplicative-expression % new-lines~opt~ format-expression

format-expression:
    range-expression
    format-expression format-operator new-lines~opt~ range-expression

range-expression:
    array-literal-expression
    range-expression .. new-lines~opt~ array-literal-expression

array-literal-expression:
    unary-expression
    unary-expression , new-lines~opt~ array-literal-expression

unary-expression:
    primary-expression
    expression-with-unary-operator

expression-with-unary-operator:
    , new-lines~opt~ unary-expression
    -not new-lines~opt~ unary-expression
    ! new-lines~opt~ unary-expression
    -bnot new-lines~opt~ unary-expression
    + new-lines~opt~ unary-expression
    dash new-lines~opt~ unary-expression
    pre-increment-expression
    pre-decrement-expression
    cast-expression
    -split new-lines~opt~ unary-expression
    -join new-lines~opt~ unary-expression

pre-increment-expression:
    ++ new-lines~opt~ unary-expression

pre-decrement-expression:
    dashdash new-lines~opt~ unary-expression

cast-expression:
    type-literal unary-expression

attributed-expression:
    type-literal variable

primary-expression:
    value
    member-access
    element-access
```

```
invocation-expression
post-increment-expression
post-decrement-expression

value:
parenthesized-expression
sub-expression
array-expression
script-block-expression
hash-literal-expression
literal
type-literal
variable

parenthesized-expression:
( new-lines~opt~ pipeline new-lines~opt~ )

sub-expression:
$( new-lines~opt~ statement-list~opt~ new-lines~opt~ )

array-expression:
@( new-lines~opt~ statement-list~opt~ new-lines~opt~ )

script-block-expression:
{ new-lines~opt~ script-block new-lines~opt~ }

hash-literal-expression:
@{ new-lines~opt~ hash-literal-body~opt~ new-lines~opt~ }

hash-literal-body:
hash-entry
hash-literal-body statement-terminators hash-entry

hash-entry:
key-expression = new-lines~opt~ statement

key-expression:
simple-name
unary-expression

post-increment-expression:
primary-expression ++

post-decrement-expression:
primary-expression dashdash

member-access: Note no whitespace is allowed after
primary-expression.
primary-expression . member-name
primary-expression :: member-name

element-access: Note no whitespace is allowed between primary-expression and
[.
primary-expression [ new-lines~opt~ expression new-lines~opt~ ]
```

```
invocation-expression: Note no whitespace is allowed after
primary-expression.

primary-expression . member-name argument-list
primary-expression :: member-name argument-list

argument-list:
( argument-expression-list~opt~ new-lines~opt~ )

argument-expression-list:
argument-expression
argument-expression new-lines~opt~ , argument-expression-list

argument-expression:
new-lines~opt~ logical-argument-expression

logical-argument-expression:
bitwise-argument-expression
logical-argument-expression -and new-lines~opt~ bitwise-argument-
expression
logical-argument-expression -or new-lines~opt~ bitwise-argument-
expression
logical-argument-expression -xor new-lines~opt~ bitwise-argument-
expression

bitwise-argument-expression:
comparison-argument-expression
bitwise-argument-expression -band new-lines~opt~ comparison-argument-
expression
bitwise-argument-expression -bor new-lines~opt~ comparison-argument-
expression
bitwise-argument-expression -bxor new-lines~opt~ comparison-argument-
expression

comparison-argument-expression:
additive-argument-expression
comparison-argument-expression comparison-operator
new-lines~opt~ additive-argument-expression

additive-argument-expression:
multiplicative-argument-expression
additive-argument-expression + new-lines~opt~ multiplicative-
argument-expression
additive-argument-expression dash new-lines~opt~ multiplicative-
argument-expression

multiplicative-argument-expression:
format-argument-expression
multiplicative-argument-expression \ new-lines~opt~ format-argument-
expression
multiplicative-argument-expression / new-lines~opt~ format-argument-
expression
multiplicative-argument-expression % new-lines~opt~ format-argument-
expression

format-argument-expression:
```

```
range-argument-expression
  format-argument-expression format-operator new-lines~opt~ range-
  argument-expression

range-argument-expression:
  unary-expression
  range-expression .. new-lines~opt~ unary-expression

member-name:
  simple-name
  string-literal
  string-literal-with-subexpression
  expression-with-unary-operator
  value

string-literal-with-subexpression:
  expandable-string-literal-with-subexpr
  expandable-here-string-literal-with-subexpr

expandable-string-literal-with-subexpr:
  expandable-string-with-subexpr-start statement-list~opt~ )
    expandable-string-with-subexpr-characters expandable-string-with-
  subexpr-end
  expandable-here-string-with-subexpr-start statement-list~opt~ )
    expandable-here-string-with-subexpr-characters
    expandable-here-string-with-subexpr-end

expandable-string-with-subexpr-characters:
  expandable-string-with-subexpr-part
  expandable-string-with-subexpr-characters expandable-string-with-
  subexpr-part

expandable-string-with-subexpr-part:
  sub-expression
  expandable-string-part

expandable-here-string-with-subexpr-characters:
  expandable-here-string-with-subexpr-part
  expandable-here-string-with-subexpr-characters expandable-here-string-
  with-subexpr-part

expandable-here-string-with-subexpr-part:
  sub-expression
  expandable-here-string-part

type-literal:
  [ type-spec ]

type-spec:
  array-type-name new-lines~opt~ dimension~opt~ ]
  generic-type-name new-lines~opt~ generic-type-arguments ]
  type-name

dimension:
  ,
```

```
dimension ,  
  
generic-type-arguments:  
    type-spec new-lines~opt~  
    generic-type-arguments , new-lines~opt~ type-spec
```

B.2.4 Attributes

Syntax

```
attribute-list:  
    attribute  
    attribute-list new-lines~opt~ attribute  
  
attribute:  
    [ new-lines~opt~ attribute-name ( attribute-arguments new-lines~opt~ )  
new-lines~opt~ ]  
    type-literal  
  
attribute-name:  
    type-spec  
  
attribute-arguments:  
    attribute-argument  
    attribute-argument new-lines~opt~ , attribute-arguments  
  
attribute-argument:  
    new-lines~opt~ expression  
    new-lines~opt~ simple-name  
    new-lines~opt~ simple-name = new-lines~opt~ expression
```

C. References

Article • 01/08/2025

Editorial note

Important

The *Windows PowerShell Language Specification 3.0* was published in December 2012 and is based on Windows PowerShell 3.0. This specification does not reflect the current state of PowerShell. There is no plan to update this documentation to reflect the current state. This documentation is presented here for historical reference.

The specification document is available as a Microsoft Word document from the Microsoft Download Center at:

<https://www.microsoft.com/download/details.aspx?id=36389> That Word document has been converted for presentation here on Microsoft Learn. During conversion, some editorial changes have been made to accommodate formatting for the Docs platform. Some typos and minor errors have been corrected.

ANSI/IEEE 754–2008, *Binary floating-point arithmetic for microprocessor systems*.

ECMA-334, *C# Language Specification*, 4th edition (June 2006), <https://www.ecma-international.org/publications-and-standards/standards/ecma-334/>. [This Ecma publication is also approved as ISO/IEC 23270:2006.]

The Open Group Base Specifications: Pattern Matching, IEEE Std 1003.1, 2004 Edition.
http://www.opengroup.org/onlinepubs/000095399/utilities/xcu_chap02.html#tag_02_13_01

The Open Group Base Specifications: Regular Expressions, IEEE Std 1003.1, 2004 Edition.
http://www.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap09.html

Ecma Technical Report TR/84, *Common Language Infrastructure (CLI) - Information Derived from Partition IV XML File*, 4th edition (June 2006), <https://www.ecma-international.org/publications-and-standards/technical-reports/ecma-tr-84/>. This TR was also published as ISO/IEC TR 23272:2006.

ISO 639-1, *Codes for the representation of names of languages - Part 1: Alpha-2 code*.

ISO 3166-1, Codes for the representation of names of countries and their subdivisions - Part 1: Country codes.

ISO/IEC 10646-1/AMD1:1996, Amendment 1 to ISO/IEC 10646-1:1993, Transformation Format for 16 planes of group 00 (UTF-16).

The Unicode Standard, Edition 5.2. The Unicode Consortium,

<http://www.unicode.org/standard/standard.html> ↗.

Windows PowerShell

Article • 09/17/2021

Updated: July 8, 2013

Windows PowerShell® is a task-based command-line shell and scripting language designed especially for system administration. Built on the .NET Framework, Windows PowerShell® helps IT professionals and power users control and automate the administration of the Windows operating system and applications that run on Windows.

The documents published here are written primarily for cmdlet, provider, and host application developers who require reference information about the APIs provided by Windows PowerShell. However, system administrators might also find the information provided by these documents useful.

For the basic information needed to start using Windows PowerShell, see [Getting Started with Windows PowerShell](#).

Windows PowerShell Documents

- [Installing the Windows PowerShell SDK](#) Provides information about how to install the Windows PowerShell SDK.
- [Writing a Windows PowerShell Module](#) Provides information for administrators, script developers, and cmdlet developers who need to package and distribute their Windows PowerShell solutions.
- [Writing a Windows PowerShell Cmdlet](#) Provides information for designing and implementing cmdlets.
- [Writing a Windows PowerShell Provider](#) Provides information for designing and implementing Windows PowerShell providers. It will help you understand how Windows PowerShell providers work, and it provides sample code that you can use to start designing or writing your own providers.
- [Writing a Windows PowerShell Host Application](#) Provides information that can be used by program managers who are designing host applications and by developers who are implementing them. The host application can define the runspace where commands are run, open sessions on a local or remote computer, and invoke the commands either synchronously or asynchronously based on the needs of the application.

- [Writing a PowerShell Formatting File](#) Provides information for the authoring of formatting files, which control the display format for the objects that are returned by commands (cmdlets, functions, and scripts).
- [Windows PowerShell Reference](#) Provides reference content for the APIs used in writing cmdlets, providers, and host applications, as well as other supporting APIs.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



[PowerShell feedback](#)

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Installing the Windows PowerShell SDK

Article • 10/23/2023

Applies To: Windows PowerShell 2.0, Windows PowerShell 3.0

The following topic describes how to install the PowerShell SDK on different versions of Windows.

Installing Windows PowerShell 3.0 SDK for Windows 8 and Windows Server 2012

Windows PowerShell 3.0 is automatically installed with Windows 8 and Windows Server 2012. In addition, you can download and install the reference assemblies for Windows PowerShell 3.0 as part of the Windows 8 SDK. These assemblies allow you to write cmdlets, providers, and host programs for Windows PowerShell 3.0. When you install the Windows SDK for Windows 8, the Windows PowerShell assemblies are automatically installed in the reference assembly folder, in `\Program Files (x86)\Reference Assemblies\Microsoft\WindowsPowerShell\3.0`. For more information, see the Windows 8 SDK download site. Windows PowerShell code samples are also available in the [powershell-sdk-samples](#) repository.

Reference assemblies

Reference assemblies are installed in the following location by default: `C:\Program Files\Reference Assemblies\Microsoft\WindowsPowerShell\V1.0`.

Note

Code that is compiled against the Windows PowerShell 2.0 assemblies cannot be loaded into Windows PowerShell 1.0 installations. However, code that is compiled against the Windows PowerShell 1.0 assemblies can be loaded into Windows PowerShell 2.0 installations.

Samples

Code samples are installed in the following location by default: `C:\Program Files\Microsoft SDKs\Windows\v7.0\Samples\sysmgmt\WindowsPowerShell\`. The following sections provide a brief description of what each sample does.

Cmdlet samples

- GetProcessSample01 - Shows how to write a simple cmdlet that gets all the processes on the local computer.
- GetProcessSample02 - Shows how to add parameters to the cmdlet. The cmdlet takes one or more process names and returns the matching processes.
- GetProcessSample03 - Shows how to add parameters that accept input from the pipeline.
- GetProcessSample04 - Shows how to handle non-terminating errors.
- GetProcessSample05 - Shows how to display a list of specified processes.
- SelectObject - Shows how to write a filter to select only certain objects.
- SelectString - Shows how to search files for specified patterns.
- StopProcessSample01 - Shows how to implement a PassThru parameter, and how to request user feedback by calls to the ShouldProcess and ShouldContinue methods. Users specify the PassThru parameter when they want to force the cmdlet to return an object,
- StopProcessSample02 - Shows how to stop a specific process.
- StopProcessSample03 - Shows how to declare aliases for parameters and how to support wildcards.
- StopProcessSample04 - Shows how to declare parameter sets, the object that the cmdlet takes as input, and how to specify the default parameter set to use.

Remoting samples

- RemoteRunspace01 - Shows how to create a remote runspace that is used to establish a remote connection.
- RemoteRunspacePool01 - Shows how to construct a remote runspace pool and how to run multiple commands concurrently by using this pool.
- Serialization01 - Shows how to look at an existing .NET class and make sure that information from selected public properties of this class is preserved across serialization/deserialization.
- Serialization02 - Shows how to look at an existing .NET class and make sure that information from instance of this class is preserved across serialization/deserialization when the information is not available in public properties of the class.
- Serialization03 - Shows how to look at an existing .NET class and make sure that instances of this class and of derived classes are deserialized (rehydrated) into live .NET objects.

Event samples

- Event01 - Shows how to create a cmdlet for event registration by deriving from ObjectEventRegistrationBase.
- Event02 - Shows how to receive notifications of Windows PowerShell events that are generated on remote computers. It uses the PSEventReceived event exposed through the Runspace class.

Hosting application samples

- Runspace01 - Shows how to use the PowerShell class to run the `Get-Process` cmdlet synchronously. The `Get-Process` cmdlet returns Process objects for each process running on the local computer.
- Runspace02 - Shows how to use the PowerShell class to run the `Get-Process` and `Sort-Object` cmdlets synchronously. The `Get-Process` cmdlet returns Process objects for each process running on the local computer, and the `Sort-Object` sorts the objects based on their Id property. The results of these commands is displayed by using a DataGridView control.
- Runspace03 - Shows how to use the PowerShell class to run a script synchronously, and how to handle non-terminating errors. The script receives a list of process names and then retrieves those processes. The results of the script, including any non-terminating errors that were generated when running the script, are displayed in a console window.
- Runspace04 - Shows how to use the PowerShell class to run commands, and how to catch terminating errors that are thrown when running the commands. Two commands are run, and the last command is passed a parameter argument that is not valid. As a result, no objects are returned and a terminating error is thrown.
- Runspace05 - Shows how to add a snap-in to an InitialSessionState object so that the cmdlet of the snap-in is available when the runspace is opened. The snap-in provides a Get-Proc cmdlet (defined by the GetProcessSample01 Sample) that is run synchronously using a PowerShell object.
- Runspace06 - Shows how to add a module to an InitialSessionState object so that the module is loaded when the runspace is opened. The module provides a Get-Proc cmdlet (defined by the GetProcessSample02 Sample) that is run synchronously using a PowerShell object.
- Runspace07 - Shows how to create a runspace, and then use that runspace to run two cmdlets synchronously using a PowerShell object.
- Runspace08 - Shows how to add commands and arguments to the pipeline of a PowerShell object and how to run the commands synchronously.
- Runspace09 - Shows how to add a script to the pipeline of a PowerShell object and how to run the script asynchronously. Events are used to handle the output of the script.

- Runspace10 - Shows how to create a default initial session state, how to add a cmdlet to the InitialSessionState, how to create a runspace that uses the initial session state, and how to run the command using a PowerShell object.
- Runspace11 - Shows how to use the ProxyCommand class to create a proxy command that calls an existing cmdlet, but restricts the set of available parameters. The proxy command is then added to an initial session state that is used to create a constrained runspace. This means that the user can access the functionality of the cmdlet only through the proxy command.
- PowerShell01 - Shows how to create a constrained runspace using an InitialSessionState object.
- PowerShell02 - Shows how to use a runspace pool to run multiple commands concurrently.

Host samples

- Host01 - Shows how to implement a host application that uses a custom host. In this sample a runspace is created that uses the custom host, and then the PowerShell API is used to run a script that calls `exit`. The host application then looks at the output of the script and prints out the results.
- Host02 - Shows how to write a host application that uses the Windows PowerShell runtime along with a custom host implementation. The host application sets the host culture to German, runs the `Get-Process` cmdlet and displays the results as you would see them by using `pwrsh.exe`, and then prints out the current data and time in German.
- Host03 - Shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console.
- Host04 - Shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console. This host application also supports displaying prompts that allow the user to specify multiple choices.
- Host05 - Shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console. This host application also supports calls to remote computers by using the `Enter-PSSession` and `Exit-PSSession` cmdlets.
- Host06 - Shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console. In addition, this sample uses the Tokenizer APIs to specify the color of the text that is entered by the user.

Provider samples

- AccessDBProviderSample01 - Shows how to declare a provider class that derives directly from the CmdletProvider class. It is included here only for completeness.
- AccessDBProviderSample02 - Shows how to overwrite the NewDrive and RemoveDrive methods to support calls to the `New-PSDrive` and `Remove-PSDrive` cmdlets. The provider class in this sample derives from the DriveCmdletProvider class.
- AccessDBProviderSample03 - Shows how to overwrite the GetItem and SetItem methods to support calls to the `Get-Item` and `Set-Item` cmdlets. The provider class in this sample derives from the ItemCmdletProvider class.
- AccessDBProviderSample04 - Shows how to overwrite container methods to support calls to the `Copy-Item`, `Get-ChildItem`, `New-Item`, and `Remove-Item` cmdlets. These methods should be implemented when the data store contains items that are containers. A container is a group of child items under a common parent item. The provider class in this sample derives from the ItemCmdletProvider class.
- AccessDBProviderSample05 - Shows how to overwrite container methods to support calls to the `Move-Item` and `Join-Path` cmdlets. These methods should be implemented when the user needs to move items within a container and if the data store contains nested containers. The provider class in this sample derives from the NavigationCmdletProvider class.
- AccessDBProviderSample06 - Shows how to overwrite content methods to support calls to the `Clear-Content`, `Get-Content`, and `Set-Content` cmdlets. These methods should be implemented when the user needs to manage the content of the items in the data store. The provider class in this sample derives from the NavigationCmdletProvider class, and it implements the `IContentCmdletProvider` interface.

Windows PowerShell Reference

Article • 09/17/2021

Windows PowerShell is a Microsoft .NET Framework-connected environment designed for administrative automation. Windows PowerShell provides a new approach to building commands, composing solutions, and creating graphical user interface-based management tools.

Windows PowerShell enables a system administrator to automate the administration of system resources by the execution of commands either directly or through scripts.

Developer Audience

The Windows PowerShell Software Development Kit (SDK) is written for command developers who require reference information about the APIs provided by Windows PowerShell. Command developers use Windows PowerShell to create both commands and providers that extend the tasks that can be performed by Windows PowerShell.

Windows PowerShell Resources

In addition to the Windows PowerShell SDK, the following resources provide more information.

[Getting Started with Windows PowerShell](#) Provides an introduction to Windows PowerShell: the language, the cmdlets, the providers, and the use of objects.

[Writing a Windows PowerShell Module](#) Provides information and examples for administrators, script developers, and cmdlet developers who need to package and distribute their Windows PowerShell solutions using Windows PowerShell modules.

[Writing a Windows PowerShell Cmdlet](#) Provides information and code examples for program managers who are designing cmdlets and for developers who are implementing cmdlet code.

[Windows PowerShell Team Blog ↗](#) The best resource for learning from and collaborating with other Windows PowerShell users. Read the Windows PowerShell Team blog, and then join the Windows PowerShell User Forum (microsoft.public.windows.powershell). Use Windows Live Search to find other Windows PowerShell blogs and resources. Then, as you develop your expertise, freely contribute your ideas.

[PowerShell module browser](#) Provides the latest versions of the command-line Help topics.

Class Libraries

[System.Management.Automation](#) This namespace is the root namespace for Windows PowerShell. It contains the classes, enumerations, and interfaces required to implement custom cmdlets. In particular, the [System.Management.Automation.Cmdlet](#) class is the base class from which all cmdlet classes must be derived. For more information about cmdlets, see.

[System.Management.Automation.Provider](#) This namespace contains the classes, enumerations, and interfaces required to implement a Windows PowerShell provider. In particular, the [System.Management.Automation.Provider.CmdletProvider](#) class is the base class from which all Windows PowerShell provider classes must be derived.

[Microsoft.PowerShell.Commands](#) This namespace contains the classes for the cmdlets and providers implemented by Windows PowerShell. Similarly, it is recommended that you create a *YourName.Commands* namespace for those cmdlets that you implement.

[System.Management.Automation.Host](#) This namespace contains the classes, enumerations, and interfaces that the cmdlet uses to define the interaction between the user and Windows PowerShell.

[System.Management.Automation.Internal](#) This namespace contains the base classes used by other namespace classes. For example, the [System.Management.Automation.Internal.CmdletMetadataAttribute](#) class is the base class for the [System.Management.CmdletAttribute](#) class.

[System.Management.Automation.Runspaces](#) This namespace contains the classes, enumerations, and interfaces used to create a Windows PowerShell runspace. In this context, the Windows PowerShell runspace is the context in which one or more Windows PowerShell pipelines invoke cmdlets. That is, cmdlets work within the context of a Windows PowerShell runspace. For more information about Windows PowerShell runspaces, see [Windows PowerShell Runspaces](#).

What's New

Article • 09/17/2021

Windows PowerShell 2.0 provides the following new features for use when writing cmdlets, providers, and host applications.

Modules

You can now package and distribute Windows PowerShell solutions by using modules. Modules allow you to partition, organize, and abstract your Windows PowerShell code into self-contained, reusable units. For more information about modules, see [Writing a Windows PowerShell Module](#).

The PowerShell class

The `PowerShell` class provides a simpler solution for creating applications, referred to as host applications, that programmatically run commands. This class allows you to create a pipeline of commands, specify the runspace that is used to run the commands, and specify invoking the commands synchronously or asynchronously.

The RunspacePool class

Runspace pools allow you to create multiple runspaces by using a single call. The `CreateRunspacePool` method provides several overloads that can be used to create runspace that have the same features, such as the same host, initial session state, and connection information.

The InitialSessionState class

The `InitialSessionState` class allows you to create a session state configuration that is used when a runspace is opened. You can create a custom configuration, a default configuration that includes the commands provided by `mshshort`, and a configuration whose commands are restricted based on the capabilities of the session.

Remote runspaces

You can now create runspace that can be opened on remote computers, allowing you to run commands on the remote machine and collect the results locally. To create a

remote runspace, you must specify information about the remote connection when creating the runspace. See the `CreateRunspace` and `CreateRunspacePool` methods for examples. The connection information is defined by the `RunspaceConnectionInfo` class.

Private runspace elements

You can now create runspaces whose elements are public or private. This allows you to create runspaces whose elements are available to the runspace, but are not available to the user. See the `ConstrainedSessionStateEntry` class to find out which elements of the runspace can be made private.

Runspace threading modes and apartment state

You can now specify how threads are created and used when running commands in a runspace. See the `System.Management.Automation.Runspaces.Runspace.ThreadOptions` and `System.Management.Automation.Runspaces.RunspacePool.ThreadOptions` properties.

You can now get the apartment state of the threads that are used to run commands in a runspace. See the `System.Management.Automation.Runspaces.Runspace.ApartmentState` and `System.Management.Automation.Runspaces.RunspacePool.ApartmentState` properties.

Transaction cmdlets

You can now create cmdlets that can be used within a transaction. When a cmdlet is used in a transaction, its actions are temporary, and they can be accepted or rejected by the transaction cmdlets provided by Windows PowerShell.

For more information about transactions, see [How to Support Transactions](#).

Transaction provider

You can now create providers that can be used within a transaction. Similar to cmdlets, when a provider is used in a transaction, its actions are temporary, and they can be accepted or rejected by the transaction cmdlets provided by Windows PowerShell.

For more information about specifying support for transaction within a provider class, see the

`System.Management.Automation.Provider.CmdletProviderAttribute.ProviderCapabilities` property.

Job cmdlets

You can now write cmdlets that can perform their action as a job. These jobs are run in the background without interacting with the current session. For more information about how Windows PowerShell supports jobs, see [Background Jobs](#).

Cmdlet output types

You can now specify the .NET Framework types that are returned by your cmdlets by declaring the `OutputType` attribute when writing your cmdlets. This will allow others to determine what type of objects are returned by a cmdlet by looking at the `OutputType` property of the cmdlet.

Event support

You can now write cmdlets that add and consume events. See the `PSEvent` class.

Proxy commands

You can now write proxy commands that can be used to run another command. A proxy command allows you to control what functionality of the source cmdlet is available to the user. For example, you can create a proxy command that removes a parameter that is supplied by the source command. See the `ProxyCommand` class.

Multiple choice prompts

You can now write applications that can provide prompts that allow the user to select multiple choices. See the `IHostUISupportsMultipleChoiceSelection` interface

Interactive sessions

You can now write applications that can start and stop an interactive session on a remote computer. See the `IHostSupportsInteractiveSession` interface.

Custom Cmdlet Help for Providers

You can now create customized Help topics for the provider cmdlets. Custom cmdlet help topics can explain how the cmdlet works in the provider path and document special features, including the dynamic parameters that the provider adds to the cmdlet.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Cmdlet Overview

Article • 09/17/2021

A cmdlet is a lightweight command that is used in the PowerShell environment. The PowerShell runtime invokes these cmdlets within the context of automation scripts that are provided at the command line. The PowerShell runtime also invokes them programmatically through PowerShell APIs.

Cmdlets

Cmdlets perform an action and typically return a Microsoft .NET object to the next command in the pipeline. A cmdlet is a single command that participates in the pipeline semantics of PowerShell. This includes binary (C#) cmdlets, advanced script functions, CDXML, and Workflows.

This SDK documentation describes how to create binary cmdlets written in C#. For information about script-based cmdlets, see:

- [about_Functions_Advanced](#)
- [about_Functions_CmdletBindingAttribute](#)
- [about_Functions_Advanced_Methods](#)

To create a binary cmdlet, you must implement a cmdlet class that derives from one of two specialized cmdlet base classes. The derived class must:

- Declare an attribute that identifies the derived class as a cmdlet.
- Define public properties that are decorated with attributes that identify the public properties as cmdlet parameters.
- Override one or more of the input processing methods to process records.

You can load the assembly that contains the class directly by using the [Import-Module](#) cmdlet, or you can create a host application that loads the assembly by using the [System.Management.Automation.Runspaces.InitialSessionState](#) API. Both methods provide programmatic and command-line access to the functionality of the cmdlet.

Cmdlet Terms

The following terms are used frequently in the PowerShell cmdlet documentation:

Cmdlet attribute

A .NET attribute that is used to declare a cmdlet class as a cmdlet. Although PowerShell uses several other attributes that are optional, the Cmdlet attribute is required. For more information about this attribute, see [Cmdlet Attribute Declaration](#).

Cmdlet parameter

The public properties that define the parameters that are available to the user or to the application that is running the cmdlet. Cmdlets can have required, named, positional, and *switch* parameters. Switch parameters allow you to define parameters that are evaluated only if the parameters are specified in the call. For more information about the different types of parameters, see [Cmdlet Parameters](#).

Parameter set

A group of parameters that can be used in the same command to perform a specific action. A cmdlet can have multiple parameter sets, but each parameter set must have at least one parameter that is unique. Good cmdlet design strongly suggests that the unique parameter also be a required parameter. For more information about parameter sets, see [Cmdlet Parameter Sets](#).

Dynamic parameter

A parameter that is added to the cmdlet at runtime. Typically, the dynamic parameters are added to the cmdlet when another parameter is set to a specific value. For more information about dynamic parameters, see [Cmdlet Dynamic Parameters](#).

Input processing methods

The [System.Management.Automation.Cmdlet](#) class provides the following virtual methods that are used to process records. All the derived cmdlet classes must override one or more of the first three methods:

- [System.Management.Automation.Cmdlet.BeginProcessing](#): Used to provide optional one-time, pre-processing functionality for the cmdlet.
- [System.Management.Automation.Cmdlet.ProcessRecord](#): Used to provide record-by-record processing functionality for the cmdlet. The [System.Management.Automation.Cmdlet.ProcessRecord](#) method might be called any number of times, or not at all, depending on the input of the cmdlet.
- [System.Management.Automation.Cmdlet.EndProcessing](#): Used to provide optional one-time, post-processing functionality for the cmdlet.

- [System.Management.Automation.Cmdlet.StopProcessing](#): Used to stop processing when the user stops the cmdlet asynchronously (for example, by pressing **CTRL + C**).

For more information about these methods, see [Cmdlet Input Processing Methods](#).

When you implement a cmdlet, you must override at least one of these input processing methods. Typically, the **ProcessRecord()** is the method that you override because it is called for every record that the cmdlet processes. In contrast, the **BeginProcessing()** method and the **EndProcessing()** method are called one time to perform pre-processing or post-processing of the records. For more information about these methods, see [Input Processing Methods](#).

ShouldProcess feature

PowerShell allows you to create cmdlets that prompt the user for feedback before the cmdlet makes a change to the system. To use this feature, the cmdlet must declare that it supports the **ShouldProcess** feature when you declare the Cmdlet attribute, and the cmdlet must call the [System.Management.Automation.Cmdlet.ShouldProcess](#) and [System.Management.Automation.Cmdlet.ShouldContinue](#) methods from within an input processing method. For more information about how to support the **ShouldProcess** functionality, see [Requesting Confirmation](#).

Transaction

A logical group of commands that are treated as a single task. The task automatically fails if any command in the group fails, and the user has the choice to accept or reject the actions performed within the transaction. To participate in a transaction, the cmdlet must declare that it supports transactions when the Cmdlet attribute is declared. Support for transactions was introduced in Windows PowerShell 2.0. For more information about transactions, see [How to Support Transactions](#).

How Cmdlets Differ from Commands

Cmdlets differ from commands in other command-shell environments in the following ways:

- Cmdlets are instances of .NET classes; they are not stand-alone executables.
- Cmdlets can be created from as few as a dozen lines of code.
- Cmdlets do not generally do their own parsing, error presentation, or output formatting. Parsing, error presentation, and output formatting are handled by the

PowerShell runtime.

- Cmdlets process input objects from the pipeline rather than from streams of text, and cmdlets typically deliver objects as output to the pipeline.
- Cmdlets are record-oriented because they process a single object at a time.

Cmdlet Base Classes

Windows PowerShell supports cmdlets that are derived from the following two base classes.

- Most cmdlets are based on .NET classes that derive from the [System.Management.Automation.Cmdlet](#) base class. Deriving from this class allows a cmdlet to use the minimum set of dependencies on the Windows PowerShell runtime. This has two benefits. The first benefit is that the cmdlet objects are smaller, and you are less likely to be affected by changes to the PowerShell runtime. The second benefit is that, if you have to, you can directly create an instance of the cmdlet object and then invoke it directly instead of invoking it through the PowerShell runtime.
- The more-complex cmdlets are based on .NET classes that derive from the [System.Management.Automation.PSCmdlet](#) base class. Deriving from this class gives you much more access to the PowerShell runtime. This access allows your cmdlet to call scripts, to access providers, and to access the current session state. (To access the current session state, you get and set session variables and preferences.) However, deriving from this class increases the size of the cmdlet object, and it means that your cmdlet is more tightly coupled to the current version of the PowerShell runtime.

In general, unless you need the extended access to the PowerShell runtime, you should derive from the [System.Management.Automation.Cmdlet](#) class. However, the PowerShell runtime has extensive logging capabilities for the execution of cmdlets. If your auditing model depends on this logging, you can prevent the execution of your cmdlet from within another cmdlet by deriving from the [System.Management.Automation.PSCmdlet](#) class.

Cmdlet Attributes

PowerShell defines several .NET attributes that are used to manage cmdlets and to specify common functionality that is provided by PowerShell and that might be required by the cmdlet. For example, attributes are used to designate a class as a cmdlet, to specify the parameters of the cmdlet, and to request the validation of input so that

cmdlet developers do not have to implement that functionality in their cmdlet code. For more information about attributes, see [PowerShell Attributes](#).

Cmdlet Names

PowerShell uses a verb-and-noun name pair to name cmdlets. For example, the `Get-Command` cmdlet included in PowerShell is used to get all the cmdlets that are registered in the command shell. The verb identifies the action that the cmdlet performs, and the noun identifies the resource on which the cmdlet performs its action.

These names are specified when the .NET class is declared as a cmdlet. For more information about how to declare a .NET class as a cmdlet, see [Cmdlet Attribute Declaration](#).

Writing Cmdlet Code

This document provides two ways to discover how cmdlet code is written. If you prefer to see the code without much explanation, see [Examples of Cmdlet Code](#). If you prefer more explanation about the code, see the [GetProc Tutorial](#), [StopProc Tutorial](#), or [SelectStr Tutorial](#) topics.

For more information about the guidelines for writing cmdlets, see [Cmdlet Development Guidelines](#).

See Also

[PowerShell Cmdlet Concepts](#)

[Writing a PowerShell Cmdlet](#)

[PowerShell SDK](#)

Windows PowerShell Cmdlet Concepts

Article • 09/17/2021

This section describes how cmdlets work.

In This Section

This section includes the following topics.

[Cmdlet Development Guidelines](#) This topic provides development guidelines that can be used to produce well-formed cmdlets.

[Cmdlet Class Declaration](#) This topic describes cmdlet class declaration.

[Approved Verbs for Windows PowerShell Commands](#) This topic lists the predefined cmdlet verbs that you can use when you declare a cmdlet class.

[Cmdlet Input Processing Methods](#) This topic describes the methods that allow a cmdlet to perform preprocessing operations, input processing operations, and post processing operations.

[Cmdlet Parameters](#) This section describes the different types of parameters that you can add to cmdlets.

[Cmdlet Attributes](#) This section describes the attributes that are used to declare .NET Framework classes as cmdlets, to declare fields as cmdlet parameters, and to declare input validation rules for parameters.

[Cmdlet Aliases](#) This topic describes cmdlet aliases.

[Cmdlet Output](#) This section describes the type of output that cmdlets can return and how to define and display the objects that are returned by cmdlets.

[Registering Cmdlets](#) This section describes how to register cmdlets by using modules and snap-ins.

[Requesting Confirmation](#) This section describes how cmdlets request confirmation from a user before they make a change to the system.

[Windows PowerShell Error Reporting](#) This section describes how cmdlets report terminating errors and non-terminating errors, and it describes how to interpret error records.

[Background Jobs](#) This topic describes how cmdlets can perform their work within background jobs that do not interfere with the commands that are executing in the current session.

[Invoking Cmdlets and Scripts Within a Cmdlet](#) This topic describes how cmdlets can invoke other cmdlets and scripts from within their input processing methods.

[Cmdlet Sets](#) This topic describes using base classes to create sets of cmdlets.

[Windows PowerShell Session State](#) This topic describes Windows PowerShell session state.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



[PowerShell feedback](#)

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Cmdlet Development Guidelines

Article • 09/17/2021

The topics in this section provide development guidelines that you can use to produce well-formed cmdlets. By leveraging the common functionality provided by the Windows PowerShell runtime and by following these guidelines, you can develop robust cmdlets with minimal effort and provide the user with a consistent experience. Additionally, you will reduce the test burden because common functionality does not require retesting.

In This Section

- [Required Development Guidelines](#)
- [Strongly Encouraged Development Guidelines](#)
- [Advisory Development Guidelines](#)

See Also

[Writing a Windows PowerShell Cmdlet](#)

[Windows PowerShell SDK](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Required Development Guidelines

Article • 09/17/2021

The following guidelines must be followed when you write your cmdlets. They are separated into guidelines for designing cmdlets and guidelines for writing your cmdlet code. If you do not follow these guidelines, your cmdlets could fail, and your users might have a poor experience when they use your cmdlets.

In this Topic

Design Guidelines

- [Use Only Approved Verbs \(RD01\)](#)
- [Cmdlet Names: Characters that cannot be Used \(RD02\)](#)
- [Parameters Names that cannot be Used \(RD03\)](#)
- [Support Confirmation Requests \(RD04\)](#)
- [Support Force Parameter for Interactive Sessions \(RD05\)](#)
- [Document Output Objects \(RD06\)](#)

Code Guidelines

- [Derive from the Cmdlet or PSCmdlet Classes \(RC01\)](#)
- [Specify the Cmdlet Attribute \(RC02\)](#)
- [Override an Input Processing Method \(RC03\)](#)
- [Specify the OutputType Attribute \(RC04\)](#)
- [Do Not Retain Handles to Output Objects \(RC05\)](#)
- [Handle Errors Robustly \(RC06\)](#)
- [Use a Windows PowerShell Module to Deploy your Cmdlets \(RC07\)](#)

Design Guidelines

The following guidelines must be followed when designing cmdlets to ensure a consistent user experience between using your cmdlets and other cmdlets. When you find a Design guideline that applies to your situation, be sure to look at the Code guidelines for similar guidelines.

Use Only Approved Verbs (RD01)

The verb specified in the Cmdlet attribute must come from the recognized set of verbs provided by Windows PowerShell. It must not be one of the prohibited synonyms. Use the constant strings that are defined by the following enumerations to specify cmdlet verbs:

- [System.Management.Automation.VerbsCommon](#)
- [System.Management.Automation.VerbsCommunications](#)
- [System.Management.Automation.VerbsData](#)
- [System.Management.Automation.VerbsDiagnostic](#)
- [System.Management.Automation.VerbsLifecycle](#)
- [System.Management.Automation.VerbsSecurity](#)
- [System.Management.Automation.VerbsOther](#)

For more information about the approved verb names, see [Cmdlet Verbs](#).

Users need a set of discoverable and expected cmdlet names. Use the appropriate verb so that the user can make a quick assessment of what a cmdlet does and to easily discover the capabilities of the system. For example, the following command-line command gets a list of all the commands on the system whose names begin with "Start": `Get-Command Start-*`. Use the nouns in your cmdlets to differentiate your cmdlets from other cmdlets. The noun indicates the resource on which the operation will be performed. The operation itself is represented by the verb.

Cmdlet Names: Characters that cannot be Used (RD02)

When you name cmdlets, do not use any of the following special characters.

[+] [Expand table](#)

Character	Name
#	number sign
,	comma
()	parentheses
{}	braces
[]	brackets
&	ampersand
-	hyphen Note: The hyphen can be used to separate the verb from the noun, but it cannot be used within the noun or within the verb.
/	slash mark
\	backslash
\$	dollar sign
^	caret
;	semicolon
:	colon
"	double quotation mark
'	single quotation mark
<>	angle brackets
	vertical bar
?	question mark
@	at sign
`	back tick (grave accent)
*	asterisk
%	percent sign
+	plus sign
=	equals sign
~	tilde

Parameters Names that cannot be Used (RD03)

Windows PowerShell provides a common set of parameters to all cmdlets plus additional parameters that are added in specific situations. When designing your own cmdlets you cannot use the following names: Confirm, Debug, ErrorAction, ErrorVariable, OutBuffer, OutVariable, WarningAction, WarningVariable, WhatIf, UseTransaction, and Verbose. For more information about these parameters, see [Common Parameter Names](#).

Support Confirmation Requests (RD04)

For cmdlets that perform an operation that modifies the system, they should call the [System.Management.Automation.Cmdlet.ShouldProcess*](#) method to request confirmation, and in special cases call the [System.Management.Automation.Cmdlet.ShouldContinue*](#) method. (The [System.Management.Automation.Cmdlet.ShouldContinue*](#) method should be called only after the [System.Management.Automation.Cmdlet.ShouldProcess*](#) method is called.)

To make these calls the cmdlet must specify that it supports confirmation requests by setting the `SupportsShouldProcess` keyword of the Cmdlet attribute. For more information about setting this attribute, see [Cmdlet Attribute Declaration](#).

Note

If the Cmdlet attribute of the cmdlet class indicates that the cmdlet supports calls to the [System.Management.Automation.Cmdlet.ShouldProcess*](#) method, and the cmdlet fails to make the call to the [System.Management.Automation.Cmdlet.ShouldProcess*](#) method, the user could modify the system unexpectedly.

Use the [System.Management.Automation.Cmdlet.ShouldProcess*](#) method for any system modification. A user preference and the `WhatIf` parameter control the [System.Management.Automation.Cmdlet.ShouldProcess*](#) method. In contrast, the [System.Management.Automation.Cmdlet.ShouldContinue*](#) call performs an additional check for potentially dangerous modifications. This method is not controlled by any user preference or the `WhatIf` parameter. If your cmdlet calls the [System.Management.Automation.Cmdlet.ShouldContinue*](#) method, it should have a `Force` parameter that bypasses the calls to these two methods and that proceeds with the operation. This is important because it allows your cmdlet to be used in non-interactive scripts and hosts.

If your cmdlets support these calls, the user can determine whether the action should actually be performed. For example, the [Stop-Process](#) cmdlet calls the [System.Management.Automation.Cmdlet.ShouldContinue*](#) method before it stops a set of critical processes, including the System, Winlogon, and Spoolsv processes.

For more information about supporting these methods, see [Requesting Confirmation](#).

Support Force Parameter for Interactive Sessions (RD05)

If your cmdlet is used interactively, always provide a Force parameter to override the interactive actions, such as prompts or reading lines of input). This is important because it allows your cmdlet to be used in non-interactive scripts and hosts. The following methods can be implemented by an interactive host.

- [System.Management.Automation.Host.PSHostUserInterface.Prompt*](#)
- [System.Management.Automation.Host.PSHostUserInterface.PromptForChoice](#)
- [System.Management.Automation.Host.IHostUISupportsMultipleChoiceSelection.PromptForChoice](#)
- [System.Management.Automation.Host.PSHostUserInterface.PromptForCredential*](#)
- [System.Management.Automation.Host.PSHostUserInterface.ReadLine*](#)
- [System.Management.Automation.Host.PSHostUserInterface.ReadLineAsSecureString*](#)

Document Output Objects (RD06)

Windows PowerShell uses the objects that are written to the pipeline. In order for users to take advantage of the objects that are returned by each cmdlet, you must document the objects that are returned, and you must document what the members of those returned objects are used for.

Code Guidelines

The following guidelines must be followed when writing cmdlet code. When you find a Code guideline that applies to your situation, be sure to look at the Design guidelines for similar guidelines.

Derive from the Cmdlet or PSCmdlet Classes (RC01)

A cmdlet must derive from either the [System.Management.Automation.Cmdlet](#) or [System.Management.Automation.PSCmdlet](#) base class. Cmdlets that derive from the [System.Management.Automation.Cmdlet](#) class do not depend on the Windows PowerShell runtime. They can be called directly from any Microsoft .NET Framework language. Cmdlets that derive from the [System.Management.Automation.PSCmdlet](#) class depend on the Windows PowerShell runtime. Therefore, they execute within a runspace.

All cmdlet classes that you implement must be public classes. For more information about these cmdlet classes, see [Cmdlet Overview](#).

Specify the Cmdlet Attribute (RC02)

For a cmdlet to be recognized by Windows PowerShell, its .NET Framework class must be decorated with the `Cmdlet` attribute. This attribute specifies the following features of the cmdlet.

- The verb-and-noun pair that identifies the cmdlet.
- The default parameter set that is used when multiple parameter sets are specified. The default parameter set is used when Windows PowerShell does not have enough information to determine which parameter set to use.
- Indicates if the cmdlet supports calls to the [System.Management.Automation.Cmdlet.ShouldProcess](#)* method. This method displays a confirmation message to the user before the cmdlet makes a change to the system. For more information about how confirmation requests are made, see [Requesting Confirmation](#).
- Indicate the impact level (or severity) of the action associated with the confirmation message. In most cases, the default value of Medium should be used. For more information about how the impact level affects the confirmation requests that are displayed to the user, see [Requesting Confirmation](#).

For more information about how to declare the cmdlet attribute, see [CmdletAttribute Declaration](#).

Override an Input Processing Method (RC03)

For the cmdlet to participate in the Windows PowerShell environment, it must override at least one of the following *input processing methods*.

[System.Management.Automation.Cmdlet.BeginProcessing](#) This method is called one time, and it is used to provide pre-processing functionality.

[System.Management.Automation.Cmdlet.ProcessRecord](#) This method is called multiple times, and it is used to provide record-by-record functionality.

[System.Management.Automation.Cmdlet.EndProcessing](#) This method is called one time, and it is used to provide post-processing functionality.

Specify the OutputType Attribute (RC04)

The `OutputType` attribute (introduced in Windows PowerShell 2.0) specifies the .NET Framework type that your cmdlet returns to the pipeline. By specifying the output type of your cmdlets you make the objects returned by your cmdlet more discoverable by other cmdlets. For more information about decorating the cmdlet class with this attribute, see [OutputType Attribute Declaration](#).

Do Not Retain Handles to Output Objects (RC05)

Your cmdlet should not retain any handles to the objects that are passed to the [System.Management.Automation.Cmdlet.WriteObject*](#) method. These objects are passed to the next cmdlet in the pipeline, or they are used by a script. If you retain the handles to the objects, two entities will own each object, which causes errors.

Handle Errors Robustly (RC06)

An administration environment inherently detects and makes important changes to the system that you are administering. Therefore, it is vital that cmdlets handle errors correctly. For more information about error records, see [Windows PowerShell Error Reporting](#).

- When an error prevents a cmdlet from continuing to process any more records, it is a terminating error. The cmdlet must call the [System.Management.Automation.Cmdlet.ThrowTerminatingError*](#) method that references an [System.Management.Automation.ErrorRecord](#) object. If an exception is not caught by the cmdlet, the Windows PowerShell runtime itself throws a terminating error that contains less information.
- For a non-terminating error that does not stop operation on the next record that is coming from the pipeline (for example, a record produced by a different process), the cmdlet must call the [System.Management.Automation.Cmdlet.WriteError*](#) method that references an [System.Management.Automation.ErrorRecord](#) object.

An example of a non-terminating error is the error that occurs if a particular process fails to stop. Calling the [System.Management.Automation.Cmdlet.WriteError](#)* method allows the user to consistently perform the actions requested and to retain the information for particular actions that fail. Your cmdlet should handle each record as independently as possible.

- The [System.Management.Automation.ErrorRecord](#) object that is referenced by the [System.Management.Automation.Cmdlet.ThrowTerminatingError](#)* and [System.Management.Automation.Cmdlet.WriteError](#)* methods requires an exception at its core. Follow the .NET Framework design guidelines when you determine the exception to use. If the error is semantically the same as an existing exception, use that exception or derive from that exception. Otherwise, derive a new exception or exception hierarchy directly from the [System.Exception](#) type.

An [System.Management.Automation.ErrorRecord](#) object also requires an error category that groups errors for the user. The user can view errors based on the category by setting the value of the `$ErrorView` shell variable to CategoryView. The possible categories are defined by the [System.Management.Automation.ErrorCategory](#) enumeration.

- If a cmdlet creates a new thread, and if the code that is running in that thread throws an unhandled exception, Windows PowerShell will not catch the error and will terminate the process.
- If an object has code in its destructor that causes an unhandled exception, Windows PowerShell will not catch the error and will terminate the process. This also occurs if an object calls Dispose methods that cause an unhandled exception.

Use a Windows PowerShell Module to Deploy your Cmdlets (RC07)

Create a Windows PowerShell module to package and deploy your cmdlets. Support for modules is introduced in Windows PowerShell 2.0. You can use the assemblies that contain your cmdlet classes directly as binary module files (this is very useful when testing your cmdlets), or you can create a module manifest that references the cmdlet assemblies. (You can also add existing snap-in assemblies when using modules.) For more information about modules, see [Writing a Windows PowerShell Module](#).

See Also

[Strongly Encouraged Development Guidelines](#)

[Advisory Development Guidelines](#)

[Writing a Windows PowerShell Cmdlet](#)

Strongly Encouraged Development Guidelines

Article • 04/11/2024

This section describes guidelines that you should follow when you write your cmdlets. They are separated into guidelines for designing cmdlets and guidelines for writing your cmdlet code. You might find that these guidelines are not applicable for every scenario. However, if they do apply and you do not follow these guidelines, your users might have a poor experience when they use your cmdlets.

Design Guidelines

The following guidelines should be followed when designing cmdlets to ensure a consistent user experience between using your cmdlets and other cmdlets. When you find a Design guideline that applies to your situation, be sure to look at the Code guidelines for similar guidelines.

Use a Specific Noun for a Cmdlet Name (SD01)

Nouns used in cmdlet naming need to be very specific so that the user can discover your cmdlets. Prefix generic nouns such as "server" with a shortened version of the product name. For example, if a noun refers to a server that is running an instance of Microsoft SQL Server, use a noun such as "SQLServer". The combination of specific nouns and the short list of approved verbs enable the user to quickly discover and anticipate functionality while avoiding duplication among cmdlet names.

To enhance the user experience, the noun that you choose for a cmdlet name should be singular. For example, use the name `Get-Process` instead of `Get-Processes`. It is best to follow this rule for all cmdlet names, even when a cmdlet is likely to act upon more than one item.

Use Pascal Case for Cmdlet Names (SD02)

Use Pascal case for parameter names. In other words, capitalize the first letter of verb and all terms used in the noun. For example, "`Clear-ItemProperty`".

Parameter Design Guidelines (SD03)

A cmdlet needs parameters that receive the data on which it must operate, and parameters that indicate information that is used to determine the characteristics of the operation. For example, a cmdlet might have a `Name` parameter that receives data from the pipeline, and the cmdlet might have a `Force` parameter to indicate that the cmdlet can be forced to perform its operation. There is no limit to the number of parameters that a cmdlet can define.

Use Standard Parameter Names

Your cmdlet should use standard parameter names so that the user can quickly determine what a particular parameter means. If a more specific name is required, use a standard parameter name, and then specify a more specific name as an alias. For example, the `Get-Service` cmdlet has a parameter that has a generic name (`Name`) and a more specific alias (`ServiceName`). Both terms can be used to specify the parameter.

For more information about parameter names and their data types, see [Cmdlet Parameter Name and Functionality Guidelines](#).

Use Singular Parameter Names

Avoid using plural names for parameters whose value is a single element. This includes parameters that take arrays or lists because the user might supply an array or list with only one element.

Plural parameter names should be used only in those cases where the value of the parameter is always a multiple-element value. In these cases, the cmdlet should verify that multiple elements are supplied, and the cmdlet should display a warning to the user if multiple elements are not supplied.

Use Pascal Case for Parameter Names

Use Pascal case for parameter names. In other words, capitalize the first letter of each word in the parameter name, including the first letter of the name. For example, the parameter name `ErrorAction` uses the correct capitalization. The following parameter names use incorrect capitalization:

- `errorAction`
- `erroraction`

Parameters That Take a List of Options

There are two ways to create a parameter whose value can be selected from a set of options.

- Define an enumeration type (or use an existing enumeration type) that specifies the valid values. Then, use the enumeration type to create a parameter of that type.
- Add the **ValidateSet** attribute to the parameter declaration. For more information about this attribute, see [ValidateSet Attribute Declaration](#).

Use Standard Types for Parameters

To ensure consistency with other cmdlets, use standard types for parameters wherever possible. For more information about the types that should be used for different parameters, see [Standard Cmdlet Parameter Names and Types](#). This topic provides links to several topics that describe the names and .NET Framework types for groups of standard parameters, such as the "activity parameters".

Use Strongly-Typed .NET Framework Types

Parameters should be defined as .NET Framework types to provide better parameter validation. For example, parameters that are restricted to one value from a set of values should be defined as an enumeration type. To support a Uniform Resource Identifier (URI) value, define the parameter as a [System.Uri](#) type. Avoid basic string parameters for all but free-form text properties.

Use Consistent Parameter Types

When the same parameter is used by multiple cmdlets, always use the same parameter type. For example, if the `Process` parameter is a [System.Int16](#) type for one cmdlet, do not make the `Process` parameter for another cmdlet a [System.UInt16](#) type.

Parameters That Take True and False

If your parameter takes only `true` and `false`, define the parameter as type [System.Management.Automation.SwitchParameter](#). A switch parameter is treated as `true` when it is specified in a command. If the parameter is not included in a command, Windows PowerShell considers the value of the parameter to be `false`. Do not define Boolean parameters.

If your parameter needs to differentiate between 3 values: \$true, \$false and "unspecified", then define a parameter of type Nullable<bool>. The need for a 3rd, "unspecified" value typically occurs when the cmdlet can modify a Boolean property of an object. In this case "unspecified" means to not change the current value of the property.

Support Arrays for Parameters

Frequently, users must perform the same operation against multiple arguments. For these users, a cmdlet should accept an array as parameter input so that a user can pass the arguments into the parameter as a Windows PowerShell variable. For example, the [Get-Process](#) cmdlet uses an array for the strings that identify the names of the processes to retrieve.

Support the PassThru Parameter

By default, many cmdlets that modify the system, such as the [Stop-Process](#) cmdlet, act as "sinks" for objects and do not return a result. These cmdlet should implement the `PassThru` parameter to force the cmdlet to return an object. When the `PassThru` parameter is specified, the cmdlet returns an object by using a call to the [System.Management.Automation.Cmdlet.WriteObject](#) method. For example, the following command stops the Calc (CalculatorApp.exe) and passes the resultant process to the pipeline.

```
PowerShell
```

```
Stop-Process -Name CalculatorApp -PassThru
```

In most cases, Add, Set, and New cmdlets should support a `PassThru` parameter.

Support Parameter Sets

A cmdlet is intended to accomplish a single purpose. However, there is frequently more than one way to describe the operation or the operation target. For example, a process might be identified by its name, by its identifier, or by a process object. The cmdlet should support all the reasonable representations of its targets. Normally, the cmdlet satisfies this requirement by specifying sets of parameters (referred to as parameter sets) that operate together. A single parameter can belong to any number of parameter sets. For more information about parameter sets, see [Cmdlet Parameter Sets](#).

When you specify parameter sets, set only one parameter in the set to `ValueFromPipeline`. For more information about how to declare the `Parameter` attribute, see [ParameterAttribute Declaration](#).

When parameter sets are used, the default parameter set is defined by the `Cmdlet` attribute. The default parameter set should include the parameters most likely to be used in an interactive Windows PowerShell session. For more information about how to declare the `Cmdlet` attribute, see [CmdletAttribute Declaration](#).

Provide Feedback to the User (SD04)

Use the guidelines in this section to provide feedback to the user. This feedback allows the user to be aware of what is occurring in the system and to make better administrative decisions.

The Windows PowerShell runtime allows a user to specify how to handle output from each call to the `Write` method by setting a preference variable. The user can set several preference variables, including a variable that determines whether the system should display information and a variable that determines whether the system should query the user before taking further action.

Support the `WriteWarning`, `WriteVerbose`, and `WriteDebug` Methods

A cmdlet should call the [System.Management.Automation.Cmdlet.WriteWarning](#) method when the cmdlet is about to perform an operation that might have an unintended result. For example, a cmdlet should call this method if the cmdlet is about to overwrite a read-only file.

A cmdlet should call the [System.Management.Automation.Cmdlet.WriteVerbose](#) method when the user requires some detail about what the cmdlet is doing. For example, a cmdlet should call this information if the cmdlet author feels that there are scenarios that might require more information about what the cmdlet is doing.

The cmdlet should call the [System.Management.Automation.Cmdlet.WriteDebug](#) method when a developer or product support engineer must understand what has corrupted the cmdlet operation. It is not necessary for the cmdlet to call the [System.Management.Automation.Cmdlet.WriteDebug](#) method in the same code that calls the [System.Management.Automation.Cmdlet.WriteVerbose](#) method because the `Debug` parameter presents both sets of information.

Support WriteProgress for Operations that take a Long Time

Cmdlet operations that take a long time to complete and that cannot run in the background should support progress reporting through periodic calls to the [System.Management.Automation.Cmdlet.WriteProgress](#) method.

Use the Host Interfaces

Occasionally, a cmdlet must communicate directly with the user instead of by using the various Write or Should methods supported by the [System.Management.Automation.Cmdlet](#) class. In this case, the cmdlet should derive from the [System.Management.Automation.PSCmdlet](#) class and use the [System.Management.Automation.PSCmdlet.Host*](#) property. This property supports different levels of communication type, including the PromptForChoice, Prompt, and WriteLine/ReadLine types. At the most specific level, it also provides ways to read and write individual keys and to deal with buffers.

Unless a cmdlet is specifically designed to generate a graphical user interface (GUI), it should not bypass the host by using the [System.Management.Automation.PSCmdlet.Host*](#) property. An example of a cmdlet that is designed to generate a GUI is the [Out-GridView](#) cmdlet.

 **Note**

Cmdlets should not use the [System.Console](#) API.

Create a Cmdlet Help File (SD05)

For each cmdlet assembly, create a Help.xml file that contains information about the cmdlet. This information includes a description of the cmdlet, descriptions of the cmdlet's parameters, examples of the cmdlet's use, and more.

Code Guidelines

The following guidelines should be followed when coding cmdlets to ensure a consistent user experience between using your cmdlets and other cmdlets. When you find a Code guideline that applies to your situation, be sure to look at the Design guidelines for similar guidelines.

Coding Parameters (SC01)

Define a parameter by declaring a public property of the cmdlet class that is decorated with the **Parameter** attribute. Parameters do not have to be static members of the derived .NET Framework class for the cmdlet. For more information about how to declare the **Parameter** attribute, see [Parameter Attribute Declaration](#).

Support Windows PowerShell Paths

The Windows PowerShell path is the mechanism for normalizing access to namespaces. When you assign a Windows PowerShell path to a parameter in the cmdlet, the user can define a custom "drive" that acts as a shortcut to a specific path. When a user designates such a drive, stored data, such as data in the Registry, can be used in a consistent way.

If your cmdlet allows the user to specify a file or a data source, it should define a parameter of type [System.String](#). If more than one drive is supported, the type should be an array. The name of the parameter should be `Path`, with an alias of `PSPath`. Additionally, the `Path` parameter should support wildcard characters. If support for wildcard characters is not required, define a `LiteralPath` parameter.

If the data that the cmdlet reads or writes has to be a file, the cmdlet should accept Windows PowerShell path input, and the cmdlet should use the [System.Management.Automation.SessionState.Path](#) property to translate the Windows PowerShell paths into paths that the file system recognizes. The specific mechanisms include the following methods:

- [System.Management.Automation.PSCmdlet.GetResolvedProviderPathFromPSPath](#)
- [System.Management.Automation.PSCmdlet.GetUnresolvedProviderPathFromPSPat h](#)
- [System.Management.Automation.PathIntrinsics.GetResolvedProviderPathFromPSPath](#)
- [System.Management.Automation.PathIntrinsics.GetUnresolvedProviderPathFromPS Path](#)

If the data that the cmdlet reads or writes is only a set of strings instead of a file, the cmdlet should use the provider content information (`Content` member) to read and write. This information is obtained from the [System.Management.Automation.Provider.CmdletProvider.InvokeProvider](#) property. These mechanisms allow other data stores to participate in the reading and writing of data.

Support Wildcard Characters

A cmdlet should support wildcard characters if possible. Support for wildcard characters occurs in many places in a cmdlet (especially when a parameter takes a string to identify one object from a set of objects). For example, the sample `Stop-Proc` cmdlet from the [StopProc Tutorial](#) defines a `Name` parameter to handle strings that represent process names. This parameter supports wildcard characters so that the user can easily specify the processes to stop.

When support for wildcard characters is available, a cmdlet operation usually produces an array. Occasionally, it does not make sense to support an array because the user might use only a single item at a time. For example, the `Set-Location` cmdlet does not need to support an array because the user is setting only a single location. In this instance, the cmdlet still supports wildcard characters, but it forces resolution to a single location.

For more information about wildcard-character patterns, see [Supporting Wildcard Characters in Cmdlet Parameters](#).

Defining Objects

This section contains guidelines for defining objects for cmdlets and for extending existing objects.

Define Standard Members

Define standard members to extend an object type in a custom `Types.ps1xml` file (use the Windows PowerShell `Types.ps1xml` file as a template). Standard members are defined by a node with the name `PSStandardMembers`. These definitions allow other cmdlets and the Windows PowerShell runtime to work with your object in a consistent way.

Define ObjectMembers to Be Used as Parameters

If you are designing an object for a cmdlet, ensure that its members map directly to the parameters of the cmdlets that will use it. This mapping allows the object to be easily sent to the pipeline and to be passed from one cmdlet to another.

Preeexisting .NET Framework objects that are returned by cmdlets are frequently missing some important or convenient members that are needed by the script developer or user. These missing members can be particularly important for display and for creating the correct member names so that the object can be correctly passed to the pipeline. Create a custom `Types.ps1xml` file to document these required members. When you

create this file, we recommend the following naming convention:

<Your_Product_Name>.Types.ps1xml.

For example, you could add a `Mode` script property to the `System.IO.FileInfo` type to display the attributes of a file more clearly. Additionally, you could add a `Count` alias property to the `System.Array` type to allow the consistent use of that property name (instead of `Length`).

Implement the `IComparable` Interface

Implement a `System.IComparable` interface on all output objects. This allows the output objects to be easily piped to various sorting and analysis cmdlets.

Update Display Information

If the display for an object does not provide the expected results, create a custom <*YourProductName*>.Format.ps1xml file for that object.

Support Well Defined Pipeline Input (SC02)

Implement for the Middle of a Pipeline

Implement a cmdlet assuming that it will be called from the middle of a pipeline (that is, other cmdlets will produce its input or consume its output). For example, you might assume that the `Get-Process` cmdlet, because it generates data, is used only as the first cmdlet in a pipeline. However, because this cmdlet is designed for the middle of a pipeline, this cmdlet allows previous cmdlets or data in the pipeline to specify the processes to retrieve.

Support Input from the Pipeline

In each parameter set for a cmdlet, include at least one parameter that supports input from the pipeline. Support for pipeline input allows the user to retrieve data or objects, to send them to the correct parameter set, and to pass the results directly to a cmdlet.

A parameter accepts input from the pipeline if the `Parameter` attribute includes the `ValueFromPipeline` keyword, the `ValueFromPipelineByPropertyName` keyword attribute, or both keywords in its declaration. If none of the parameters in a parameter set support the `ValueFromPipeline` or `ValueFromPipelineByPropertyName` keywords, the cmdlet

cannot meaningfully be placed after another cmdlet because it will ignore any pipeline input.

Support the ProcessRecord Method

To accept all the records from the preceding cmdlet in the pipeline, your cmdlet must implement the [System.Management.Automation.Cmdlet.ProcessRecord](#) method. Windows PowerShell calls this method multiple times, once for every record that is sent to your cmdlet.

Write Single Records to the Pipeline (SC03)

When a cmdlet returns objects, the cmdlet should write the objects immediately as they are generated. The cmdlet should not hold them in order to buffer them into a combined array. The cmdlets that receive the objects as input will then be able to process, display, or process and display the output objects without delay. A cmdlet that generates output objects one at a time should call the [System.Management.Automation.Cmdlet.WriteObject](#) method. A cmdlet that generates output objects in batches (for example, because an underlying API returns an array of output objects) should call the [System.Management.Automation.Cmdlet.WriteObject](#) Method with its second parameter set to `true`.

Make Cmdlets Case-Insensitive and Case-Preserving (SC04)

By default, Windows PowerShell itself is case-insensitive. However, because it deals with many preexisting systems, Windows PowerShell does preserve case for ease of operation and compatibility. In other words, if a character is supplied in uppercase letters, Windows PowerShell keeps it in uppercase letters. For systems to work well, a cmdlet needs to follow this convention. If possible, it should operate in a case-insensitive way. It should, however, preserve the original case for cmdlets that occur later in a command or in the pipeline.

See Also

[Required Development Guidelines](#)

[Advisory Development Guidelines](#)

[Writing a Windows PowerShell Cmdlet](#)

Advisory Development Guidelines

Article • 05/22/2025

This section describes guidelines that you should consider to ensure good development and user experiences. Sometimes they might apply, and sometimes they might not.

Design Guidelines

The following guidelines should be considered when designing cmdlets. When you find a Design guideline that applies to your situation, be sure to look at the Code guidelines for similar guidelines.

Support an InputObject Parameter (AD01)

Because Windows PowerShell works directly with Microsoft .NET Framework objects, a .NET Framework object is often available that exactly matches the type the user needs to perform a particular operation. `InputObject` is the standard name for a parameter that takes such an object as input. For example, the sample `Stop-Proc` cmdlet in the [StopProc Tutorial](#) defines an `InputObject` parameter of type `Process` that supports the input from the pipeline. The user can get a set of process objects, manipulate them to select the exact objects to stop, and then pass them to the `Stop-Proc` cmdlet directly.

Support the Force Parameter (AD02)

Occasionally, a cmdlet needs to protect the user from performing a requested operation. Such a cmdlet should support a `Force` parameter to allow the user to override that protection if the user has permissions to perform the operation.

For example, the `Remove-Item` cmdlet doesn't normally remove a read-only file. However, this cmdlet supports a `Force` parameter so a user can force removal of a read-only file. If the user already has permission to modify the read-only attribute, and the user removes the file, use of the `Force` parameter simplifies the operation. However, if the user doesn't have permission to remove the file, the `Force` parameter has no effect.

Handle Credentials Through Windows PowerShell (AD03)

A cmdlet should define a `Credential` parameter to represent credentials. This parameter must be of type `System.Management.Automation.PSCredential` and must be defined using a Credential attribute declaration. This support automatically prompts the user for the user

name, for the password, or for both when a full credential isn't supplied directly. For more information about the Credential attribute, see [Credential Attribute Declaration](#).

Support Encoding Parameters (AD04)

If your cmdlet reads or writes text to or from a binary form, such as writing to or reading from a file in a filesystem, then your cmdlet has to have Encoding parameter that specifies how the text is encoded in the binary form.

Test Cmdlets Should Return a Boolean (AD05)

Cmdlets that perform tests against their resources should return a [System.Boolean](#) type to the pipeline so that they can be used in conditional expressions.

Code Guidelines

The following guidelines should be considered when writing cmdlet code. When you find a guideline that applies to your situation, be sure to look at the Design guidelines for similar guidelines.

Follow Cmdlet Class Naming Conventions (AC01)

By following standard naming conventions, you make your cmdlets more discoverable, and you help the user understand exactly what the cmdlets do. This practice is particularly important for other developers using Windows PowerShell because cmdlets are public types.

Define a Cmdlet in the Correct Namespace

You normally define the class for a cmdlet in a .NET Framework namespace that appends `.Commands` to the namespace that represents the product in which the cmdlet runs. For example, cmdlets that are included with Windows PowerShell are defined in the `Microsoft.PowerShell.Commands` namespace.

Name the Cmdlet Class to Match the Cmdlet Name

When you name the .NET Framework class that implements a cmdlet, name the class `<Verb>` `<Noun>Command`, where you replace the `<Verb>` and `<Noun>` placeholders with the verb and noun used for the cmdlet name. For example, the [Get-Process](#) cmdlet is implemented by a class called `GetProcessCommand`.

If No Pipeline Input Override the BeginProcessing Method (AC02)

If your cmdlet doesn't accept input from the pipeline, processing should be implemented in the [System.Management.Automation.Cmdlet.BeginProcessing](#) method. Use of this method allows Windows PowerShell to maintain ordering between cmdlets. The first cmdlet in the pipeline always returns its objects before the remaining cmdlets in the pipeline get a chance to start their processing.

To Handle Stop Requests Override the StopProcessing Method (AC03)

Override the [System.Management.Automation.Cmdlet.StopProcessing](#) method so that your cmdlet can handle stop signal. Some cmdlets take a long time to complete their operation, and they let a long time pass between calls to the Windows PowerShell runtime, such as when the cmdlet blocks the thread in long-running RPC calls. This includes cmdlets that make calls to the [System.Management.Automation.Cmdlet.WriteObject](#) method, to the [System.Management.Automation.Cmdlet.LogError](#) method, and to other feedback mechanisms that may take a long time to complete. For these cases the user might need to send a stop signal to these cmdlets.

Implement the IDisposable Interface (AC04)

If your cmdlet has objects that aren't disposed of (written to the pipeline) by the [System.Management.Automation.Cmdlet.ProcessRecord](#) method, your cmdlet might require additional object disposal. For example, if your cmdlet opens a file handle in its [System.Management.Automation.Cmdlet.BeginProcessing](#) method and keeps the handle open for use by the [System.Management.Automation.Cmdlet.ProcessRecord](#) method, this handle has to be closed at the end of processing.

The Windows PowerShell runtime doesn't always call the [System.Management.Automation.Cmdlet.EndProcessing](#) method. For example, the [System.Management.Automation.Cmdlet.EndProcessing](#) method might not be called if the cmdlet is canceled midway through its operation or if a terminating error occurs in any part of the cmdlet. Therefore, the .NET Framework class for a cmdlet that requires object cleanup should implement the complete [System.IDisposable](#) interface pattern, including the finalizer, so that the Windows PowerShell runtime can call both the [System.Management.Automation.Cmdlet.EndProcessing](#) and [System.IDisposable.Dispose*](#) methods at the end of processing.

Use Serialization-friendly Parameter Types (AC05)

To support running your cmdlet on remote computers, use types that can be serialized on the client computer and then rehydrated on the server computer. The following types are serialization-friendly.

Primitive types:

- Byte, SByte, Decimal, Single, Double, Int16, Int32, Int64, UInt16, UInt32, and UInt64.
- Boolean, Guid, Byte[], TimeSpan, DateTime, Uri, and Version.
- Char, String, XmlDocument.

Built-in rehydratable types:

- PSPrimitiveDictionary
- SwitchParameter
- PSListModifier
- PSCredential
- IPAddress, MailAddress
- CultureInfo
- X509Certificate2, X500DistinguishedName
- DirectorySecurity, FileSecurity, RegistrySecurity

Other types:

- SecureString
- Containers (lists and dictionaries of the above type)

Use SecureString for Sensitive Data (AC06)

When handling sensitive data always use the [System.Security.SecureString](#) data type. This could include pipeline input to parameters, as well as returning sensitive data to the pipeline.

While .NET recommends against using **SecureString** for new development, PowerShell continues to support the **SecureString** class for backward compatibility. Using a **SecureString** is still more secure than using a plain text string. PowerShell still relies on the **SecureString** type to avoid accidentally exposing the contents to the console or in logs. Use **SecureString** carefully, because it can be easily converted to a plain text string. For a full discussion about using **SecureString**, see the [System.Security.SecureString class](#) documentation.

See Also

[Required Development Guidelines](#)

Strongly Encouraged Development Guidelines

Writing a Windows PowerShell Cmdlet

Cmdlet Class Declaration

Article • 09/17/2021

A Microsoft .NET Framework class is declared as a cmdlet by specifying the **Cmdlet** attribute as metadata for the class. (The **Cmdlet** attribute is the only required attribute for all cmdlets). When you specify the **Cmdlet** attribute, you must specify the verb-and-noun pair that identifies the cmdlet to the user. And, you must describe the Windows PowerShell functionality that the cmdlet supports. For more information about the declaration syntax that is used to specify the **Cmdlet** attribute, see [Cmdlet Attribute Declaration](#).

ⓘ Note

The **Cmdlet** attribute is defined by the **System.Management.Automation.CmdletAttribute** class. The properties of this class correspond to the declaration parameters that are used when you declare the attribute.

Nouns

The noun of the cmdlet specifies the resources upon which the cmdlet acts. The noun differentiates your cmdlets from other cmdlets.

Nouns in cmdlet names must be specific, and in the case of generic nouns, such as *server*, it is best to add a short prefix that differentiates your resource from other similar resources. For example, a cmdlet name that includes a noun with a prefix is `Get-SQLServer`. The combination of a specific noun with a more general verb enables the user to quickly locate the cmdlet by its action and then identify the cmdlet by its resource while avoiding unnecessary cmdlet name duplication.

For a list of special characters that cannot be used in cmdlet names, see [Required Development Guidelines](#).

Verbs

When you specify a verb, the development guidelines require you to use one of the predefined verbs provided by Windows PowerShell. By using one of these predefined verbs, you will ensure consistency between the cmdlets that you write and the cmdlets

that are written by Microsoft and by others. For example, the "Get" verb is used for cmdlets that retrieve data.

For more information about guidelines for verbs, see [Cmdlet Verb Names](#). For a list of special characters that cannot be used in cmdlet names, see [Required Development Guidelines](#).

Supporting Windows PowerShell Functionality

The **Cmdlet** attribute also allows you to specify that your cmdlet supports some of the common functionality that is provided by Windows PowerShell. This includes support for common functionality such as user feedback confirmation (referred to as support for the `ShouldProcess` feature) and support for transactions. (Support for transactions was introduced in Windows PowerShell 2.0).

For more information about the declaration syntax that is used to specify the **Cmdlet** attribute, see [Cmdlet Attribute Declaration](#).

Cmdlet Class Definition

The following code is the definition for a `GetProc` cmdlet class. Notice that Pascal casing is used and that the name of the class includes the verb and noun of the cmdlet.

```
C#  
  
[Cmdlet(VerbsCommon.Get, "Proc")]
public class GetProcCommand : Cmdlet
```

Pascal Casing

When you name cmdlets, use Pascal casing. For example, the `Get-Item` and `Get-ItemProperty` cmdlets show the correct way to use capitalization when you are naming cmdlets.

See Also

[System.Management.Automation.CmdletAttribute](#)

[CmdletAttribute Declaration](#)

[Cmdlet Verb Names](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Approved Verbs for PowerShell Commands

Article • 04/29/2025

PowerShell uses a verb-noun pair for the names of cmdlets and for their derived .NET classes. The verb part of the name identifies the action that the cmdlet performs. The noun part of the name identifies the entity on which the action is performed. For example, the `Get-Command` cmdlet retrieves all the commands that are registered in PowerShell.

ⓘ Note

PowerShell uses the term *verb* to describe a word that implies an action even if that word isn't a standard verb in the English language. For example, the term `New` is a valid PowerShell verb name because it implies an action even though it isn't a verb in the English language.

Each approved verb has a corresponding *alias prefix* defined. We use this alias prefix in aliases for commands using that verb. For example, the alias prefix for `Import` is `ip` and, accordingly, the alias for `Import-Module` is `ipmo`. This is a recommendation but not a rule; in particular, it need not be respected for command aliases mimicking well known commands from other environments.

Verb Naming Recommendations

The following recommendations help you choose an appropriate verb for your cmdlet, to ensure consistency between the cmdlets that you create, the cmdlets that are provided by PowerShell, and the cmdlets that are designed by others.

- Use one of the predefined verb names provided by PowerShell
- Use the verb to describe the general scope of the action, and use parameters to further refine the action of the cmdlet.
- Don't use a synonym of an approved verb. For example, always use `Remove`, never use `Delete` or `Eliminate`.
- Use only the form of each verb that's listed in this topic. For example, use `Get`, but don't use `Getting` or `Gets`.
- Don't use the following reserved verbs or aliases. The PowerShell language and a rare few cmdlets use these verbs under exceptional circumstances.
 - `ForEach` (`foreach`)
 - `Ping` (`pi`)
 - `Sort` (`sr`)

- o `Tee` (`te`)
- o `Where` (`wh`)

You may get a complete list of verbs using the `Get-Verb` cmdlet.

Similar Verbs for Different Actions

The following similar verbs represent different actions.

`New` VS. `Add`

Use the `New` verb to create a new resource. Use the `Add` to add something to an existing container or resource. For example, `Add-Content` adds output to an existing file.

`New` VS. `Set`

Use the `New` verb to create a new resource. Use the `Set` verb to modify an existing resource, optionally creating it if it doesn't exist, such as the `Set-Variable` cmdlet.

`Find` VS. `Search`

Use the `Find` verb to look for an object. Use the `Search` verb to create a reference to a resource in a container.

`Get` VS. `Read`

Use the `Get` verb to obtain information about a resource (such as a file) or to obtain an object with which you can access the resource in future. Use the `Read` verb to open a resource and extract information contained within.

`Invoke` VS. `Start`

Use the `Invoke` verb to perform synchronous operations, such as running a command and waiting for it to end. Use the `Start` verb to begin asynchronous operations, such as starting an autonomous process.

`Ping` VS. `Test`

Use the `Test` verb.

Common Verbs

PowerShell uses the [System.Management.Automation.VerbsCommon](#) enumeration class to define generic actions that can apply to almost any cmdlet. The following table lists most of the defined verbs.

 Expand table

Verb (alias)	Action	Synonyms to avoid
Add (a)	Adds a resource to a container, or attaches an item to another item. For example, the <code>Add-Content</code> cmdlet adds content to a file. This verb is paired with <code>Remove</code> .	Append, Attach, Concatenate, Insert
Clear (cl)	Removes all the resources from a container but doesn't delete the container. For example, the <code>Clear-Content</code> cmdlet removes the contents of a file but doesn't delete the file.	Flush, Erase, Release, Unmark, Unset, Nullify
Close (cs)	Changes the state of a resource to make it inaccessible, unavailable, or unusable. This verb is paired with <code>Open</code> .	
Copy (cp)	Copies a resource to another name or to another container. For example, the <code>Copy-Item</code> cmdlet copies an item (such as a file) from one location in the data store to another location.	Duplicate, Clone, Replicate, Sync
Enter (et)	Specifies an action that allows the user to move into a resource. For example, the <code>Enter-PSSession</code> cmdlet places the user in an interactive session. This verb is paired with <code>Exit</code> .	Push, Into
Exit (ex)	Sets the current environment or context to the most recently used context. For example, the <code>Exit-PSSession</code> cmdlet places the user in the session that was used to start the interactive session. This verb is paired with <code>Enter</code> .	Pop, Out
Find (fd)	Looks for an object in a container that's unknown, implied, optional, or specified.	Search
Format (f)	Arranges objects in a specified form or layout	
Get (g)	Specifies an action that retrieves a resource. This verb is paired with <code>Set</code> .	Read, Open, Cat, Type, Dir, Obtain, Dump, Acquire, Examine, Find, Search
Hide (h)	Makes a resource undetectable. For example, a cmdlet whose name includes the Hide verb might conceal a service from a user. This verb is paired with <code>Show</code> .	Block

Verb (alias)	Action	Synonyms to avoid
Join (j)	Combines resources into one resource. For example, the <code>Join-Path</code> cmdlet combines a path with one of its child paths to create a single path. This verb is paired with <code>Split</code> .	Combine, Unite, Connect, Associate
Lock (lk)	Secures a resource. This verb is paired with <code>Unlock</code> .	Restrict, Secure
Move (m)	Moves a resource from one location to another. For example, the <code>Move-Item</code> cmdlet moves an item from one location in the data store to another location.	Transfer, Name, Migrate
New (n)	Creates a resource. (The <code>Set</code> verb can also be used when creating a resource that includes data, such as the <code>Set-Variable</code> cmdlet.)	Create, Generate, Build, Make, Allocate
Open (op)	Changes the state of a resource to make it accessible, available, or usable. This verb is paired with <code>Close</code> .	
Optimize (om)	Increases the effectiveness of a resource.	
Pop (pop)	Removes an item from the top of a stack. For example, the <code>Pop-Location</code> cmdlet changes the current location to the location that was most recently pushed onto the stack.	
Push (pu)	Adds an item to the top of a stack. For example, the <code>Push-Location</code> cmdlet pushes the current location onto the stack.	
Redo (re)	Resets a resource to the state that was undone.	
Remove (r)	Deletes a resource from a container. For example, the <code>Remove-Variable</code> cmdlet deletes a variable and its value. This verb is paired with <code>Add</code> .	Clear, Cut, Dispose, Discard, Erase
Rename (rn)	Changes the name of a resource. For example, the <code>Rename-Item</code> cmdlet, which is used to access stored data, changes the name of an item in the data store.	Change
Reset (rs)	Sets a resource back to its original state.	
Resize(rz)	Changes the size of a resource.	
Search (sr)	Creates a reference to a resource in a container.	Find, Locate
Select (sc)	Locates a resource in a container. For example, the <code>Select-String</code> cmdlet finds text in strings and files.	Find, Locate

Verb (alias)	Action	Synonyms to avoid
Set (s)	Replaces data on an existing resource or creates a resource that contains some data. For example, the Set-Date cmdlet changes the system time on the local computer. (The New verb can also be used to create a resource.) This verb is paired with Get.	Write, Reset, Assign, Configure, Update
Show (sh)	Makes a resource visible to the user. This verb is paired with Hide.	Display, Produce
Skip (sk)	Bypasses one or more resources or points in a sequence.	Bypass, Jump
Split (s1)	Separates parts of a resource. For example, the Split-Path cmdlet returns different parts of a path. This verb is paired with Join.	Separate
Step (st)	Moves to the next point or resource in a sequence.	
Switch (sw)	Specifies an action that alternates between two resources, such as to change between two locations, responsibilities, or states.	
Undo (un)	Sets a resource to its previous state.	
Unlock (uk)	Releases a resource that was locked. This verb is paired with Lock.	Release, Unrestrict, Unsecure
Watch (wc)	Continually inspects or monitors a resource for changes.	

Communications Verbs

PowerShell uses the [System.Management.Automation.VerbsCommunications](#) class to define actions that apply to communications. The following table lists most of the defined verbs.

[\[+\] Expand table](#)

Verb (alias)	Action	Synonyms to avoid
Connect (cc)	Creates a link between a source and a destination. This verb is paired with Disconnect.	Join, Telnet, Login
Disconnect (dc)	Breaks the link between a source and a destination. This verb is paired with Connect.	Break, Logoff
Read (rd)	Acquires information from a source. This verb is paired with Write.	Acquire, Prompt, Get
Receive (rc)	Accepts information sent from a source. This verb is paired with Read.	Read, Accept, Peek

Verb (alias)	Action	Synonyms to avoid
	Send .	
Send (sd)	Delivers information to a destination. This verb is paired with Receive .	Put, Broadcast, Mail, Fax
Write (wr)	Adds information to a target. This verb is paired with Read .	Put, Print

Data Verbs

PowerShell uses the [System.Management.Automation.VerbsData](#) class to define actions that apply to data handling. The following table lists most of the defined verbs.

[\[+\] Expand table](#)

Verb Name (alias)	Action	Synonyms to avoid
Backup (ba)	Stores data by replicating it.	Save, Burn, Replicate, Sync
Checkpoint (ch)	Creates a snapshot of the current state of the data or of its configuration.	Diff
Compare (cr)	Evaluates the data from one resource against the data from another resource.	Diff
Compress (cm)	Compacts the data of a resource. Pairs with Expand .	Compact
Convert (cv)	Changes the data from one representation to another when the cmdlet supports bidirectional conversion or when the cmdlet supports conversion between multiple data types.	Change, Resize, Resample
ConvertFrom (cf)	Converts one primary type of input (the cmdlet noun indicates the input) to one or more supported output types.	Export, Output, Out
ConvertTo (ct)	Converts from one or more types of input to a primary output type (the cmdlet noun indicates the output type).	Import, Input, In
Dismount (dm)	Detaches a named entity from a location. This verb is paired with Mount .	Unmount, Unlink
Edit (ed)	Modifies existing data by adding or removing content.	Change, Update, Modify
Expand (en)	Restores the data of a resource that has been compressed to its original state. This verb is paired with Compress .	Explode, Uncompress

Verb Name (alias)	Action	Synonyms to avoid
Export (ep)	Encapsulates the primary input into a persistent data store, such as a file, or into an interchange format. This verb is paired with Import.	Extract, Backup
Group (gp)	Arranges or associates one or more resources	
Import (ip)	Creates a resource from data that's stored in a persistent data store (such as a file) or in an interchange format. For example, the Import-Csv cmdlet imports data from a comma-separated value (.csv) file to objects that can be used by other cmdlets. This verb is paired with Export.	BulkLoad, Load
Initialize (in)	Prepares a resource for use, and sets it to a default state.	Erase, Init, Renew, Rebuild, Reinitialize, Setup
Limit (l)	Applies constraints to a resource.	Quota
Merge (mg)	Creates a single resource from multiple resources.	Combine, Join
Mount (mt)	Attaches a named entity to a location. This verb is paired with Dismount.	Connect
Out (o)	Sends data out of the environment. For example, the Out-Printer cmdlet sends data to a printer.	
Publish (pb)	Makes a resource available to others. This verb is paired with Unpublish.	Deploy, Release, Install
Restore (rr)	Sets a resource to a predefined state, such as a state set by Checkpoint. For example, the Restore-Computer cmdlet starts a system restore on the local computer.	Repair, Return, Undo, Fix
Save (sv)	Preserves data to avoid loss.	
Sync (sy)	Assures that two or more resources are in the same state.	Replicate, Coerce, Match
Unpublish (ub)	Makes a resource unavailable to others. This verb is paired with Publish.	Uninstall, Revert, Hide
Update (ud)	Brings a resource up-to-date to maintain its state, accuracy, conformance, or compliance. For example, the Update-FormatData cmdlet updates and adds formatting files to the current PowerShell console.	Refresh, Renew, Recalculate, Re-index

Diagnostic Verbs

PowerShell uses the [System.Management.Automation.VerbsDiagnostic](#) class to define actions that apply to diagnostics. The following table lists most of the defined verbs.

[+] [Expand table](#)

Verb (alias)	Action	Synonyms to avoid
Debug (db)	Examines a resource to diagnose operational problems.	Diagnose
Measure (ms)	Identifies resources that are consumed by a specified operation, or retrieves statistics about a resource.	Calculate, Determine, Analyze
Ping (pi)	Deprecated - Use the Test verb instead.	
Repair (rp)	Restores a resource to a usable condition	Fix, Restore
Resolve (rv)	Maps a shorthand representation of a resource to a more complete representation.	Expand, Determine
Test (t)	Verifies the operation or consistency of a resource.	Diagnose, Analyze, Salvage, Verify
Trace (tr)	Tracks the activities of a resource.	Track, Follow, Inspect, Dig

Lifecycle Verbs

PowerShell uses the [System.Management.Automation.VerbsLifecycle](#) class to define actions that apply to the lifecycle of a resource. The following table lists most of the defined verbs.

[+] [Expand table](#)

Verb (alias)	Action	Synonyms to avoid
Approve (ap)	Confirms or agrees to the status of a resource or process.	
Assert (as)	Affirms the state of a resource.	Certify
Build (bd)	Creates an artifact (usually a binary or document) out of some set of input files (usually source code or declarative documents.) This verb was added in PowerShell 6.	

Verb (alias)	Action	Synonyms to avoid
Complete (cp)	Concludes an operation.	
Confirm (cn)	Acknowledges, verifies, or validates the state of a resource or process.	Acknowledge, Agree, Certify, Validate, Verify
Deny (dn)	Refuses, objects, blocks, or opposes the state of a resource or process.	Block, Object, Refuse, Reject
Deploy (dp)	Sends an application, website, or solution to a remote target[s] in such a way that a consumer of that solution can access it after deployment is complete. This verb was added in PowerShell 6.	
Disable (d)	Configures a resource to an unavailable or inactive state. For example, the <code>Disable-PSBreakpoint</code> cmdlet makes a breakpoint inactive. This verb is paired with <code>Enable</code> .	Halt, Hide
Enable (e)	Configures a resource to an available or active state. For example, the <code>Enable-PSBreakpoint</code> cmdlet makes a breakpoint active. This verb is paired with <code>Disable</code> .	Start, Begin
Install (is)	Places a resource in a location, and optionally initializes it. This verb is paired with <code>Uninstall</code> .	Setup
Invoke (i)	Performs an action, such as running a command or a method.	Run, Start
Register (rg)	Creates an entry for a resource in a repository such as a database. This verb is paired with <code>Unregister</code> .	
Request (rq)	Asks for a resource or asks for permissions.	
Restart (rt)	Stops an operation and then starts it again. For example, the <code>Restart-Service</code> cmdlet stops and then starts a service.	Recycle
Resume (ru)	Starts an operation that has been suspended. For example, the <code>Resume-Service</code> cmdlet starts a service that has been suspended. This verb is paired with <code>Suspend</code> .	
Start (sa)	Initiates an operation. For example, the <code>Start-Service</code> cmdlet starts a service. This verb is paired with <code>Stop</code> .	Launch, Initiate, Boot
Stop (sp)	Discontinues an activity. This verb is paired with <code>Start</code> .	End, Kill, Terminate, Cancel
Submit (sb)	Presents a resource for approval.	Post

Verb (alias)	Action	Synonyms to avoid
Suspend (ss)	Pauses an activity. For example, the <code>Suspend-Service</code> cmdlet pauses a service. This verb is paired with <code>Resume</code> .	Pause
Uninstall (us)	Removes a resource from an indicated location. This verb is paired with <code>Install</code> .	
Unregister (ur)	Removes the entry for a resource from a repository. This verb is paired with <code>Register</code> .	Remove
Wait (w)	Pauses an operation until a specified event occurs. For example, the <code>Wait-Job</code> cmdlet pauses operations until one or more of the background jobs are complete.	Sleep, Pause

Security Verbs

PowerShell uses the `System.Management.Automation.VerbsSecurity` class to define actions that apply to security. The following table lists most of the defined verbs.

[] Expand table

Verb (alias)	Action	Synonyms to avoid
Block (bl)	Restricts access to a resource. This verb is paired with <code>Unblock</code> .	Prevent, Limit, Deny
Grant (gr)	Allows access to a resource. This verb is paired with <code>Revoke</code> .	Allow, Enable
Protect (pt)	Safeguards a resource from attack or loss. This verb is paired with <code>Unprotect</code> .	Encrypt, Safeguard, Seal
Revoke (rk)	Specifies an action that doesn't allow access to a resource. This verb is paired with <code>Grant</code> .	Remove, Disable
Unblock (ul)	Removes restrictions to a resource. This verb is paired with <code>Block</code> .	Clear, Allow
Unprotect (up)	Removes safeguards from a resource that were added to prevent it from attack or loss. This verb is paired with <code>Protect</code> .	Decrypt, Unseal

Other Verbs

PowerShell uses the [System.Management.Automation.VerbsOther](#) class to define canonical verb names that don't fit into a specific verb name category such as the common, communications, data, lifecycle, or security verb names verbs.

 Expand table

Verb (alias)	Action	Synonyms to avoid
Use (u)	Uses or includes a resource to do something.	

See Also

- [System.Management.Automation.VerbsCommon](#)
- [System.Management.Automation.VerbsCommunications](#)
- [System.Management.Automation.VerbsData](#)
- [System.Management.Automation.VerbsDiagnostic](#)
- [System.Management.Automation.VerbsLifecycle](#)
- [System.Management.Automation.VerbsSecurity](#)
- [System.Management.Automation.VerbsOther](#)
- [Cmdlet Declaration](#)
- [Windows PowerShell Programmer's Guide](#)
- [Windows PowerShell Shell SDK](#)

Cmdlet Input Processing Methods

Article • 09/17/2021

Cmdlets must override one or more of the input processing methods described in this topic to perform their work. These methods allow the cmdlet to perform operations of pre-processing, input processing, and post-processing. These methods also allow you to stop cmdlet processing. For a more detailed example of how to use these methods, see [SelectStr Tutorial](#).

Pre-Processing Operations

Cmdlets should override the [System.Management.Automation.Cmdlet.BeginProcessing](#) method to add any preprocessing operations that are valid for all the records that will be processed later by the cmdlet. When PowerShell processes a command pipeline, PowerShell calls this method once for each instance of the cmdlet in the pipeline. For more information about how PowerShell invokes the command pipeline, see [Cmdlet Processing Lifecycle](#).

The following code shows an implementation of the BeginProcessing method.

```
C#  
  
protected override void BeginProcessing()  
{  
    // Replace the WriteObject method with the logic required by your cmdlet.  
    WriteObject("This is a test of the BeginProcessing template.");  
}
```

Input Processing Operations

Cmdlets can override the [System.Management.Automation.Cmdlet.ProcessRecord](#) method to process the input that is sent to the cmdlet. When PowerShell processes a command pipeline, PowerShell calls this method for each input record that is processed by the cmdlet. For more information about how PowerShell invokes the command pipeline, see [Cmdlet Processing Lifecycle](#).

The following code shows an implementation of the ProcessRecord method.

```
C#  
  
protected override void ProcessRecord()  
{
```

```
// Replace the WriteObject method with the logic required by your cmdlet.  
WriteObject("This is a test of the ProcessRecord template.");  
}
```

Post-Processing Operations

Cmdlets should override the [System.Management.Automation.Cmdlet.EndProcessing](#) method to add any post-processing operations that are valid for all the records that were processed by the cmdlet. For example, your cmdlet might have to clean up object variables after it is finished processing.

When PowerShell processes a command pipeline, PowerShell calls this method once for each instance of the cmdlet in the pipeline. However, it is important to remember that the PowerShell runtime will not call the EndProcessing method if the cmdlet is canceled midway through its input processing or if a terminating error occurs in any part of the cmdlet. For this reason, a cmdlet that requires object cleanup should implement the complete [System.IDisposable](#) pattern, including a finalizer, so that the runtime can call both the EndProcessing and [System.IDisposable.Dispose](#) methods at the end of processing. For more information about how PowerShell invokes the command pipeline, see [Cmdlet Processing Lifecycle](#).

The following code shows an implementation of the EndProcessing method.

C#

```
protected override void EndProcessing()  
{  
    // Replace the WriteObject method with the logic required by your cmdlet.  
    WriteObject("This is a test of the EndProcessing template.");  
}
```

See Also

[System.Management.Automation.Cmdlet.BeginProcessing](#)

[System.Management.Automation.Cmdlet.ProcessRecord](#)

[System.Management.Automation.Cmdlet.EndProcessing](#)

[SelectStr Tutorial](#)

[System.IDisposable](#)

[Windows PowerShell Shell SDK](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Cmdlet Parameters

Article • 09/17/2021

Cmdlet parameters provide the mechanism that allows a cmdlet to accept input. Parameters can accept input directly from the command line, or from objects passed to the cmdlet through the pipeline. The arguments (also known as *values*) of these parameters can specify the input that the cmdlet accepts, how the cmdlet should perform its actions, and the data that the cmdlet returns to the pipeline.

In This Section

[Declaring Properties as Parameters](#) Provides basic information you must understand before you declare the parameters of a cmdlet.

[Types of Cmdlet Parameters](#) Describes the different types of parameters that you can declare in cmdlets.

[Cmdlet Parameter Name and Functionality Guidelines](#) Discusses the names, recommended data type, and functionality of standard parameters.

[Parameter Aliases](#) Discusses the aliases that you can define for parameters.

[Common Parameter Names](#) This topic describes the parameters that Windows PowerShell adds to cmdlets.

[Cmdlet Parameter Sets](#) Discusses how parameter sets enable you to write a single cmdlet that can perform different actions for different scenarios.

[Cmdlet Dynamic Parameters](#) Discusses parameters that are available to the user under special conditions.

[Supporting Wildcard Characters in Cmdlet Parameters](#) Describes how to provide support for wildcard characters when you design a cmdlet that will be run against a group of resources.

[Validating Parameter Input](#) Describes how Windows PowerShell validates the arguments passed to cmdlet parameters.

[Input Filter Parameters](#) Discusses the `Filter`, `Include`, and `Exclude` parameters that filter the set of input objects that the cmdlet affects.

Related Sections

See Also

[Parameter Attribute Declaration](#)

[Windows PowerShell Cmdlets](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Declaring Properties as Parameters

Article • 09/17/2021

This topic provides basic information you must understand before you declare the parameters of a cmdlet.

To declare the parameters of a cmdlet within your cmdlet class, define the public properties that represent each parameter, and then add one or more Parameter attributes to each property. The Windows PowerShell runtime uses the Parameter attributes to identify the property as a cmdlet parameter. The basic syntax for declaring the Parameter attribute is `[Parameter()]`.

Here is an example of a property defined as a required parameter.

C#

```
[Parameter(Position = 0, Mandatory = true)]
public string UserName
{
    get { return userName; }
    set { userName = value; }
}
private string userName;
```

Here are some things to remember about parameters.

- A parameter must be explicitly marked as public. Parameters that are not marked as public default to internal and will not be found by the Windows PowerShell runtime.
- Parameters should be defined as Microsoft .NET Framework types to provide better parameter validation. For example, parameters that are restricted to one value out of a set of values should be defined as an enumeration type. Parameters that take a Uniform Resource Identifier (URI) value should be of type `System.Uri`.
- Avoid basic string parameters for all but free-form text properties.
- You can add a parameter to any number of parameter sets. For more information about parameter sets, see [Cmdlet Parameter Sets](#).

Windows PowerShell also provides a set of common parameters that are automatically available to every cmdlet. For more information about these parameters and their aliases, see [Cmdlet Common Parameters](#).

See Also

[Cmdlet Common Parameters](#)

[Types of Cmdlet Parameter](#)

[Writing a Windows PowerShell Cmdlet](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Types of Cmdlet Parameters

Article • 02/25/2025

This topic describes the different types of parameters that you can declare in cmdlets. Cmdlet parameters can be positional, named, required, optional, or switch parameters.

Positional and Named Parameters

All cmdlet parameters are either named or positional parameters. A named parameter requires that you type the parameter name and argument when calling the cmdlet. A positional parameter requires only that you type the arguments in relative order. The system then maps the first unnamed argument to the first positional parameter. The system maps the second unnamed argument to the second unnamed parameter, and so on. By default, all cmdlet parameters are named parameters.

To define a named parameter, omit the `Position` keyword in the `Parameter` attribute declaration, as shown in the following parameter declaration.

```
C#  
  
[Parameter(ValueFromPipeline=true)]  
public string UserName  
{  
    get { return userName; }  
    set { userName = value; }  
}  
private string userName;
```

To define a positional parameter, add the `Position` keyword in the `Parameter` attribute declaration, and then specify a position. In the following sample, the `UserName` parameter is declared as a positional parameter with position 0. This means that the first argument of the call is automatically bound to this parameter.

```
C#  
  
[Parameter(Position = 0)]  
public string UserName  
{  
    get { return userName; }  
    set { userName = value; }  
}  
private string userName;
```

Note

Good cmdlet design recommends that the most-used parameters be declared as positional parameters so that the user doesn't have to enter the parameter name when the cmdlet is run.

Positional and named parameters accept single arguments or multiple arguments separated by commas. Multiple arguments are allowed only if the parameter accepts a collection such as an array of strings. You may mix positional and named parameters in the same cmdlet. In this case, the system retrieves the named arguments first, and then attempts to map the remaining unnamed arguments to the positional parameters.

The following commands show the different ways in which you can specify single and multiple arguments for the parameters of the `Get-Command` cmdlet. Notice that in the last two samples, `-Name` doesn't need to be specified because the `Name` parameter is defined as a positional parameter.

PowerShell

```
Get-Command -Name Get-Service
Get-Command -Name Get-Service,Set-Service
Get-Command Get-Service
Get-Command Get-Service,Set-Service
```

Mandatory and Optional Parameters

You can also define cmdlet parameters as mandatory or optional parameters. (A mandatory parameter must be specified before the PowerShell runtime invokes the cmdlet.) By default, parameters are defined as optional.

To define a mandatory parameter, add the `Mandatory` keyword in the Parameter attribute declaration, and set it to `true`, as shown in the following parameter declaration.

C#

```
[Parameter(Position = 0, Mandatory = true)]
public string UserName
{
    get { return userName; }
    set { userName = value; }
```

```
}
```

```
private string userName;
```

To define an optional parameter, omit the `Mandatory` keyword in the `Parameter` attribute declaration, as shown in the following parameter declaration.

C#

```
[Parameter(Position = 0)]
public string UserName
{
    get { return userName; }
    set { userName = value; }
}
private string userName;
```

Switch Parameters

PowerShell provides a `System.Management.Automation.SwitchParameter` type that allows you to define a parameter whose default value `false` unless the parameter is specified when the cmdlet is called. Whenever possible, use switch parameters instead of Boolean parameters.

Consider the following example. Many PowerShell cmdlets return output. However, these cmdlets have a `PassThru` switch parameter that overrides the default behavior. When you use the `PassThru` parameter, the cmdlet returns output objects to the pipeline.

To define a switch parameter, declare the property as the `[SwitchParameter]` type, as shown in the following sample.

C#

```
[Parameter()]
public SwitchParameter GoodBye
{
    get { return goodbye; }
    set { goodbye = value; }
}
private bool goodbye;
```

To make the cmdlet act on the parameter when it's specified, use the following structure within one of the input processing methods.

C#

```
protected override void ProcessRecord()
{
    WriteObject("Switch parameter test: " + userName + ".");
    if (goodbye)
    {
        WriteObject(" Goodbye!");
    }
} // End ProcessRecord
```

By default, switch parameters are excluded from positional parameters. You *can* override that in the **Parameter** attribute, but it can confuse users.

Design switch parameters so that using parameter changes the default behavior of the command to a less common or more complicated mode. The simplest behavior of a command should be the default behavior that doesn't require the use of switch parameters. Base the behavior controlled by the switch on the value of the switch, not the presence of the parameter.

There are several ways to test for the presence of a switch parameters:

- `Invocation.BoundParameters` contains the switch parameter name as a key
- `PSCmdlet.ParameterSetName` when the switch defines a unique parameter set

For example, it's possible to provide an explicit value for the switch using `-MySwitch:$false` or splatting. If you only test for the presence of the parameter, the command behaves as if the switch value is `$true` instead of `$false`.

See Also

[Writing a Windows PowerShell Cmdlet](#)

Standard Cmdlet Parameter Names and Types

Article • 09/17/2021

Cmdlet parameter names should be consistent across the cmdlets that you design. The following topics list the parameter names that we recommend you use when you declare cmdlet parameters. The topics also describe the recommended data type and functionality of each parameter.

In This Section

[Activity Parameters](#)

[Date and Time Parameters](#)

[Format Parameters](#)

[Property Parameters](#)

[Quantity Parameters](#)

[Resource Parameters](#)

[Security Parameters](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Activity Parameters

Article • 06/23/2022

The following table lists the recommended names and functionality for activity parameters.

[Expand table](#)

Parameter	Functionality
Append Data type: SwitchParameter	Implement this parameter so that the user can add content to the end of a resource when the parameter is specified.
CaseSensitive Data type: SwitchParameter	Implement this parameter so the user can require case sensitivity when the parameter is specified.
Command Data type: String	Implement this parameter so the user can specify a command string to run.
CompatibleVersion Data type: System.Version object	Implement this parameter so the user can specify the semantics that the cmdlet must be compatible with for compatibility with previous versions.
Compress Data type: SwitchParameter	Implement this parameter so that data compression is used when the parameter is specified.
Compress Data type: Keyword	Implement this parameter so that the user can specify the algorithm to use for data compression.
Continuous Data type: SwitchParameter	Implement this parameter so that data is processed until the user terminates the cmdlet when the parameter is specified. If the parameter is not specified, the cmdlet processes a predefined amount of data and then terminates the operation.
Create Data type: SwitchParameter	Implement this parameter to indicate that a resource is created if one does not already exist when the parameter is specified.
Delete Data type: SwitchParameter	Implement this parameter so that resources are deleted when the cmdlet has completed its operation when the parameter is specified.
Drain Data type: SwitchParameter	Implement this parameter to indicate that outstanding work items are processed before the cmdlet processes new data when the parameter is

Parameter	Functionality
	specified. If the parameter is not specified, the work items are processed immediately.
Erase Data type: Int32	Implement this parameter so that the user can specify the number of times a resource is erased before it is deleted.
ErrorLevel Data type: Int32	Implement this parameter so that the user can specify the level of errors to report.
Exclude Data type: String[]	Implement this parameter so that the user can exclude something from an activity. For more information about how to use input filters, see Input Filter Parameters .
Filter Data type: Keyword	Implement this parameter so that the user can specify a filter that selects the resources upon which to perform the cmdlet action. For more information about how to use input filters, see Input Filter Parameters .
Follow Data type: SwitchParameter	Implement this parameter so that progress is tracked when the parameter is specified.
Force Data type: SwitchParameter	Implement this parameter to indicate that the user can perform an action even if restrictions are encountered when the parameter is specified. The parameter does not allow security to be compromised. For example, this parameter lets a user overwrite a read-only file.
Include Data type: String[]	Implement this parameter so that the user can include something in an activity. For more information about how to use input filters, see Input Filter Parameters .
Incremental Data type: SwitchParameter	Implement this parameter to indicate that processing is performed incrementally when the parameter is specified. For example, this parameter lets a user perform incremental backups that back up files only since the last backup.
InputObject Data type: Object	Implement this parameter when the cmdlet takes input from other cmdlets. When you define an InputObject parameter, always specify the ValueFromPipeline keyword when you declare the Parameter attribute. For more information about using input filters, see Input Filter Parameters .
Insert Data type: SwitchParameter	Implement this parameter so that the cmdlet inserts an item when the parameter is specified.
Interactive Data type: SwitchParameter	Implement this parameter so that the cmdlet works interactively with the user when the parameter is specified.

Parameter	Functionality
Interval Data type: HashTable	Implement this parameter so that the user can specify a hash table of keywords that contains the values. The following example shows sample values for the Interval parameter: <code>-interval @{ResumeScan=15; Retry=3}</code> .
Log Data type: SwitchParameter	Implement this parameter audit the actions of the cmdlet when the parameter is specified.
NoClobber Data type: SwitchParameter	Implement this parameter so that the resource will not be overwritten when the parameter is specified. This parameter generally applies to cmdlets that create new objects so that they can be prevented from overwriting existing objects with the same name.
Notify Data type: SwitchParameter	Implement this parameter so that the user will be notified that the activity is complete when the parameter is specified.
NotifyAddress Data type: Email address	Implement this parameter so that the user can specify the e-mail address to use to send a notification when the Notify parameter is specified.
Overwrite Data type: SwitchParameter	Implement this parameter so that the cmdlet overwrites any existing data when the parameter is specified.
Prompt Data type: String	Implement this parameter so that the user can specify a prompt for the cmdlet.
Quiet Data type: SwitchParameter	Implement this parameter so that the cmdlet suppresses user feedback during its actions when the parameter is specified.
Recurse Data type: SwitchParameter	Implement this parameter so that the cmdlet recursively performs its actions on resources when the parameter is specified.
Repair Data type: SwitchParameter	Implement this parameter so that the cmdlet will attempt to correct something from a broken state when the parameter is specified.
RepairString Data type: String	Implement this parameter so that the user can specify a string to use when the Repair parameter is specified.
Retry Data type: Int32	Implement this parameter so the user can specify the number of times the cmdlet will attempt an action.
Select Data type: Keyword array	Implement this parameter so that the user can specify an array of the types of items.

Parameter	Functionality
Stream Data type: SwitchParameter	Implement this parameter so the user can stream multiple output objects through the pipeline when the parameter is specified.
Strict Data type: SwitchParameter	Implement this parameter so that all errors are handled as terminating errors when the parameter is specified.
TempLocation Data type: String	Implement this parameter so the user can specify the location of temporary data that is used during the operation of the cmdlet.
Timeout Data type: Int32	Implement this parameter so that the user can specify the timeout interval (in milliseconds).
Truncate Data type: SwitchParameter	Implement this parameter so that the cmdlet will truncate its actions when the parameter is specified. If the parameter is not specified, the cmdlet performs another action.
Verify Data type: SwitchParameter	Implement this parameter so that the cmdlet will test to determine whether an action has occurred when the parameter is specified.
Wait Data type: SwitchParameter	Implement this parameter so that the cmdlet will wait for user input before continuing when the parameter is specified.
WaitTime Data type: Int32	Implement this parameter so that the user can specify the duration (in seconds) that the cmdlet will wait for user input when the Wait parameter is specified.

See Also

[Cmdlet Parameters](#)

[Writing a Windows PowerShell Cmdlet](#)

[Windows PowerShell SDK](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Date and Time Parameters

Article • 09/17/2021

The following table lists recommended names and functionality for parameters that handle date and time information. Date and time parameters are typically used to record when something is created or accessed.

[+] Expand table

Parameter	Functionality
Accessed Data type: SwitchParameter	Implement this parameter so that when it is specified the cmdlet will operate on the resources that have been accessed based on the date and time specified by the Before and After parameters. If this parameter is specified, the Created and Modified parameters must not be specified.
After Data type: DateTime	Implement this parameter to specify the date and time after which the cmdlet was used. For the After parameter to work, the cmdlet must also have an Accessed , Created , or Modified parameter. And, that parameter must be set to true when the cmdlet is called.
Before Data type: DateTime	Implement this parameter to specify the date and time before which the cmdlet was used. For the Before parameter to work, the cmdlet must also have an Accessed , Created , or Modified parameter. And, that parameter must be set to true when the cmdlet is called.
Created Data type: SwitchParameter	Implement this parameter so that when it is specified the cmdlet will operate on the resources that have been created based on the date and time specified by the Before and After parameters. If this parameter is specified, the Accessed and Modified parameters must not be specified.
Exact Data type: SwitchParameter	Implement this parameter so that when it is specified the resource term must match the resource name exactly. When the parameter is not specified the resource term and name do not need to match exactly.
Modified Data type: DateTime	Implement this parameter so that when it is specified the cmdlet will operate on resources that have been changed based on the date and time specified by the Before and After parameters. If this parameter is specified, the Accessed and Created parameters must not be specified.

See Also

[Cmdlet Parameters](#)

[Writing a Windows PowerShell Cmdlet](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Format Parameters

Article • 09/17/2021

The following table lists recommended names and functionality for parameters that are used to format or to generate data.

 Expand table

Parameter	Functionality
As Data type: Keyword	Implement this parameter to specify the cmdlet output format. For example, possible values could be Text or Script.
Binary Data type: SwitchParameter	Implement this parameter to indicate that the cmdlet handles binary values.
Encoding Data type: Keyword	Implement this parameter to specify the type of encoding that is supported. For example, possible values could be ASCII, UTF8, Unicode, UTF7, BigEndianUnicode, Byte, and String.
NewLine Data type: SwitchParameter	Implement this parameter so that the newline characters are supported when the parameter is specified.
ShortName Data type: SwitchParameter	Implement this parameter so that short names are supported when the parameter is specified.
Width Data type: Int32	Implement this parameter so that the user can specify the width of the output device.
Wrap Data type: SwitchParameter	Implement this parameter so that text wrapping is supported when the parameter is specified.

See Also

[Cmdlet Parameters](#)

[Writing a Windows PowerShell Cmdlet](#)

[Windows PowerShell SDK](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Property Parameters

Article • 09/17/2021

The following table lists the recommended names and functionality for property parameters.

[Expand table](#)

Parameter	Functionality
Count Data type: Int32	Implement this parameter so that the user can specify the number of objects to be processed.
Description Data type: String	Implement this parameter so that the user can specify a description for a resource.
From Data type: String	Implement this parameter so that the user can specify the reference object to get information from.
Id Data type: Resource dependent	Implement this parameter so that the user can specify the identifier of a resource.
Input Data type: String	Implement this parameter so that the user can specify the input file specification.
Location Data type: String	Implement this parameter so that the user can specify the location of the resource.
LogName Data type: String	Implement this parameter so that the user can specify the name of the log file to process or use.
Name Data type: String	Implement this parameter so that the user can specify the name of the resource.
Output Data type: String	Implement this parameter so that the user can specify the output file.
Owner Data type: String	Implement this parameter so that the user can specify the name of the owner of the resource.
Property Data type: String	Implement this parameter so that the user can specify the name or the names of the properties to use.
Reason Data type: String	Implement this parameter so that the user can specify why this cmdlet is being invoked.
Regex Data type:	Implement this parameter so that regular expressions are used when the parameter is specified. When this parameter is specified, wildcard

Parameter	Functionality
SwitchParameter	characters are not resolved.
Speed Data type: Int32	Implement this parameter so that the user can specify the baud rate. The user sets this parameter to the speed of the resource.
State Data type: Keyword array	Implement this parameter so that the user can specify the names of states, such as KEYDOWN.
Value Data type: Object	Implement this parameter so that the user can specify a value to provide to the cmdlet.
Version Data type: String	Implement this parameter so that the user can specify the version of the property.

See Also

[Cmdlet Parameters](#)

[Writing a Windows PowerShell Cmdlet](#)

[Windows PowerShell SDK](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Quantity Parameters

Article • 09/17/2021

The following table lists the recommended names and functionality for quantity parameters.

[Expand table](#)

Parameter	Functionality
All Data type: Boolean	Implement this parameter so that <code>true</code> indicates that all resources should be acted upon instead of a default subset of resources. Implement this parameter so that <code>false</code> indicates a subset of the resources.
Allocation Data type: Int32	Implement this parameter so that the user can specify the number of items to allocate.
BlockCount Data type: Int64	Implement this parameter so that the user can specify the block count.
Count Data type: Int64	Implement this parameter so that the user can specify the count.
Scope Data type: Keyword	Implement this parameter so that the user can specify the scope to operate on.

See Also

[Cmdlet Parameters](#)

[Writing a Windows PowerShell Cmdlet](#)

[Windows PowerShell SDK](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

Resource Parameters

Article • 09/17/2021

The following table lists the recommended names and functionality for resource parameters. For these parameters, the resources could be the assembly that contains the cmdlet class or the host application that is running the cmdlet.

[\[+\] Expand table](#)

Parameter	Functionality
Application Data type: String	Implement this parameter so that the user can specify an application.
Assembly Data type: String	Implement this parameter so that the user can specify an assembly.
Attribute Data type: String	Implement this parameter so that the user can specify an attribute.
Class Data type: String	Implement this parameter so that the user can specify a Microsoft .NET Framework class.
Cluster Data type: String	Implement this parameter so that the user can specify a cluster.
Culture Data type: String	Implement this parameter so that the user can specify the culture in which to run the cmdlet.
Domain Data type: String	Implement this parameter so that the user can specify the domain name.
Drive Data type: String	Implement this parameter so that the user can specify a drive name.
Event Data type: String	Implement this parameter so that the user can specify an event name.

Parameter	Functionality
Interface Data type: String	Implement this parameter so that the user can specify a network interface name.
IpAddress Data type: String	Implement this parameter so that the user can specify an IP address.
Job Data type: String	Implement this parameter so that the user can specify a job.
LiteralPath Data type: String	Implement this parameter so that the user can specify the path to a resource when wildcard characters are not supported. (Use the Path parameter when wildcard characters are supported.)
Mac Data type: String	Implement this parameter so that the user can specify a media access controller (MAC) address.
ParentId Data type: String	Implement this parameter so that the user can specify the parent identifier.
Path Data type: String, String[]	Implement this parameter so that the user can indicate the paths to a resource when wildcard characters are supported. (Use the LiteralPath parameter when wildcard characters are not supported.) We recommend that you develop this parameter so that it supports the full <code>provider:path</code> syntax used by providers. We also recommend that you develop it so that it works with as many providers as possible.
Port Data type: Integer, String	Implement this parameter so that the user can specify an integer value for networking or a string value such as "biztalk" for other types of port.
Printer Data type: Integer, String	Implement this parameter so that the user can specify the printer for the cmdlet to use.
Size Data type: Int32	Implement this parameter so that the user can specify a size.
TID Data type: String	Implement this parameter so that the user can specify a transaction identifier (TID) for the cmdlet.

Parameter	Functionality
Type Data type: String	Implement this parameter so that the user can specify the type of resource on which to operate.
URL Data type: String	Implement this parameter so that the user can specify a Uniform Resource Locator (URL).
User Data type: String	Implement this parameter so that the user can specify their name or the name of another user.

See Also

[Cmdlet Parameters](#)

[Writing a Windows PowerShell Cmdlet](#)

[Windows PowerShell SDK](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Security Parameters

Article • 10/13/2023

The following table lists the recommended names and functionality for parameters used to provide security information for an operation, such as parameters that specify certificate key and privilege information.

[\[\] Expand table](#)

Parameter	Functionality
ACL Data type: String	Implement this parameter to specify the access control level of protection for a catalog or for a Uniform Resource Identifier (URI).
CertFile Data type: String	Implement this parameter so that the user can specify the name of a file that contains one of the following: <ul style="list-style-type: none">- A Base64 or Distinguished Encoding Rules (DER) encoded x.509 certificate- A Public Key Cryptography Standards (PKCS) #12 file that contains at least one certificate and key
CertIssuerName Data type: String	Implement this parameter so that the user can specify the name of the issuer of a certificate or so that the user can specify a substring.
CertRequestFile Data type: String	Implement this parameter to specify the name of a file that contains a Base64 or DER-encoded PKCS #10 certificate request.
CertSerialNumber Data type: String	Implement this parameter to specify the serial number that was issued by the certification authority.
CertStoreLocation Data type: String	Implement this parameter so that the user can specify the location of the certificate store. The location is typically a file path.
CertSubjectName Data type: String	Implement this parameter so that the user can specify the issuer of a certificate or so that the user can specify a substring.
CertUsage Data type: String	Implement this parameter to specify the key usage or the enhanced key usage. The key can be represented as a bit mask, a bit, an object identifier (OID), or a string.

Parameter	Functionality
Credential Data type: <code>System.Management.Automation.PSCredential</code>	Implement this parameter so that the cmdlet will automatically prompt the user for a user name or password. A prompt for both is displayed if a full credential is not supplied directly.
CSPName Data type: String	Implement this parameter so that the user can specify the name of the certificate service provider (CSP).
CSPType Data type: Integer	Implement this parameter so that the user can specify the type of CSP.
Group Data type: String	Implement this parameter so that the user can specify a collection of principals for access. For more information, see the description of the Principal parameter.
KeyAlgorithm Data type: String	Implement this parameter so that the user can specify the key generation algorithm to use for security.
KeyContainerName Data type: String	Implement this parameter so that the user can specify the name of the key container.
KeyLength Data type: Integer	Implement this parameter so that the user can specify the length of the key in bits.
Operation Data type: String	Implement this parameter so that the user can specify an action that can be performed on a protected object.
Principal Data type: String	Implement this parameter so that the user can specify a unique identifiable entity for access.
Privilege Data type: String, String[]	Implement this parameter so that the user can specify the rights a cmdlet needs to perform an operation for a particular entity.
Role Data type: String	Implement this parameter so that the user can specify a set of operations that can be performed by an entity.
SaveCred Data type: SwitchParameter	Implement this parameter so that credentials that were previously saved by the user will be used when the parameter is specified.
Scope Data type: String	Implement this parameter so that the user can specify the group of protected objects for the

Parameter	Functionality
	cmdlet.
SID Data type: String	Implement this parameter so that the user can specify a unique identifier that represents a principal.
Trusted Data type: SwitchParameter	Implement this parameter so that trust levels are supported when the parameter is specified.
TrustLevel Data type: Keyword	Implement this parameter so that the user can specify the trust level that is supported. For example, possible values include internet, intranet, and fulltrust.

See Also

[Cmdlet Parameters](#)

[Writing a Windows PowerShell Cmdlet](#)

[Windows PowerShell SDK](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Parameter Aliases

Article • 09/17/2021

Cmdlet parameters can also have aliases. You can use the aliases instead of the parameter names when you type or specify the parameter in a command.

Benefits of Using Aliases

Adding aliases to parameters provides the following benefits.

- You can provide a shortcut so that the user does not have to use the complete parameter name when the cmdlet is called. For example, you could use the "CN" alias instead of the parameter name "ComputerName".
- You can define multiple aliases if you want to provide different names for the same parameter. You might want to define multiple aliases if you have to work with multiple user groups that refer to the same data in different ways.
- You can provide backwards compatibility for existing scripts if the name of a parameter changes.
- By using the Alias attribute along with the ValueFromPipelineByName attribute, you can define a parameter that allows your cmdlet to bind to different object types. For example, say you had two objects of different types and the first object had a writer property and the second object had an editor property. If your cmdlet had a parameter that had writer and editor aliases and the cmdlet accepted pipeline input based in property names, your cmdlet could bind to both objects using the two parameter aliases.

For more information about aliases that can be used with specific parameters, see [Common Parameter Names](#).

Defining Parameter Aliases

To define an alias for a parameter, declare the Alias attribute, as shown in the following parameter declaration. In this example, multiple aliases are defined for the same parameter. (For more information, see [How to Declare Cmdlet Parameters](#).)

C#

```
[Alias("UN","Writer","Editor")]
[Parameter()]
```

```
public string UserName
{
    get { return userName; }
    set { userName = value; }
}
private string userName;
```

See Also

[Common Parameter Names](#)

[How to Declare Cmdlet Parameters](#)

[Writing a Windows PowerShell Cmdlet](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Common Parameter Names

Article • 10/04/2023

The parameters described in this topic are referred to as **common parameters**. They're added to cmdlets by the PowerShell runtime and can't be declared by the cmdlet.

ⓘ Note

These parameters are also added to provider cmdlets and to functions that are decorated with the `CmdletBinding` attribute.

General Common Parameters

The following parameters are added to all cmdlets and can be accessed whenever the cmdlet is run. These parameters are defined by the [CommonParameters](#) class.

Debug (alias: db)

Data type: [SwitchParameter](#)

This parameter specifies whether programmer-level debugging messages that can be displayed at the command line. These messages are intended for troubleshooting the operation of the cmdlet, and are generated by calls to the [WriteDebug](#) method. Debug messages don't need to be localized.

ErrorAction (alias: ea)

Data type: [Enumeration](#)

This parameter specifies what action should take place when an error occurs. The possible values for this parameter are defined by the [ActionPreference](#) enumeration.

ErrorVariable (alias: ev)

Data type: [String](#)

This parameter specifies the variable in which to place objects when an error occurs. To append to this variable, use `+varname` rather than clearing and setting the variable.

InformationAction (alias: inf)

Data type: Enumeration

This parameter specifies what action should take place when output is sent to the **Information** stream. The possible values for this parameter are defined by the [ActionPreference](#) enumeration.

InformationVariable (alias: iv)

Data type: String

This parameter specifies the variable in which to save output objects written to the **Information** stream. To append to this variable, use `+varname` rather than clearing and setting the variable.

OutBuffer (alias: ob)

Data type: Int32

This parameter defines the number of objects to store in the output buffer before any objects are passed down the pipeline. By default, objects are passed immediately down the pipeline.

OutVariable (alias: ov)

Data type: String

This parameter specifies the variable in which to place all output objects generated by the cmdlet. To append to this variable, use `+varname` rather than clearing and setting the variable.

PipelineVariable (alias: pv)

Data type: String

This parameter stores the value of the current pipeline element as a variable for any named command as it flows through the pipeline.

ProgressAction (alias: proga)

Data type: Enumeration

Determines how PowerShell responds to progress updates generated by a script, cmdlet, or provider, such as the progress bars generated by the `Write-Progress` cmdlet.

This parameter was added in PowerShell 7.4.

Verbose (alias: vb)

Data type: `SwitchParameter`

This parameter specifies whether the cmdlet writes explanatory messages that can be displayed at the command line. These messages are intended to provide additional help to the user, and are generated by calls to the [WriteVerbose](#) method.

WarningAction (alias: wa)

Data type: `Enumeration`

This parameter specifies what action should take place when the cmdlet writes a warning message. The possible values for this parameter are defined by the [ActionPreference](#) enumeration.

WarningVariable (alias: wv)

Data type: `String`

This parameter specifies the variable in which warning messages can be saved. To append to this variable, use `+varname` rather than clearing and setting the variable.

Risk-Mitigation Parameters

The following parameters are added to cmdlets that requests confirmation before they perform their action. For more information about confirmation requests, see [Requesting Confirmation](#). These parameters are defined by the [ShouldProcessParameters](#) class.

Confirm (alias: cf)

Data type: `SwitchParameter`

This parameter specifies whether the cmdlet displays a prompt that asks if the user is sure that they want to continue.

WhatIf (alias: wi)

Data type: **SwitchParameter**

This parameter specifies whether the cmdlet writes a message that describes the effects of running the cmdlet without actually performing any action.

Transaction Parameters

The following parameter is added to cmdlets that support transactions. These parameters are defined by the [TransactionParameters](#) class.

Transaction support was introduced in PowerShell 3.0 and discontinued in PowerShell 6.0.

UseTransaction (alias: usetx)

Data type: **SwitchParameter**

This parameter specifies whether the cmdlet uses the current transaction to perform its action.

See Also

- [about_CommonParameters](#)
- [System.Management.Automation.Internal.CommonParameters](#)
- [System.Management.Automation.Internal.ShouldProcessParameters](#)
- [System.Management.Automation.Internal.TransactionParameters](#)
- [Writing a Windows PowerShell Cmdlet](#)
- [Windows PowerShell SDK](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Cmdlet parameter sets

Article • 09/17/2021

PowerShell uses parameter sets to enable you to write a single cmdlet that can do different actions for different scenarios. Parameter sets enable you to expose different parameters to the user. And, to return different information based on the parameters specified by the user.

Examples of parameter sets

For example, the PowerShell `Get-EventLog` cmdlet returns different information depending on whether the user specifies the `List` or `LogName` parameter. If the `List` parameter is specified, the cmdlet returns information about the log files themselves but not the event information they contain. If the `LogName` parameter is specified, the cmdlet returns information about the events in a specific event log. The `List` and `LogName` parameters identify two separate parameter sets.

Unique parameter

Each parameter set must have a unique parameter that the PowerShell runtime uses to expose the appropriate parameter set. If possible, the unique parameter should be a mandatory parameter. When a parameter is mandatory, the user must specify the parameter, and the PowerShell runtime uses that parameter to identify the parameter set. The unique parameter can't be mandatory if your cmdlet is designed to run without specifying any parameters.

Multiple parameter sets

In the following illustration, the left column shows three valid parameter sets. **Parameter A** is unique to the first parameter set, **parameter B** is unique to the second parameter set, and **parameter C** is unique to the third parameter set. In the right column, the parameter sets don't have a unique parameter.

Valid Parameter Sets	Invalid Parameter Sets
A D	A B
B D E	B C
C D	A C

Parameter set requirements

The following requirements apply to all parameter sets.

- Each parameter set must have at least one unique parameter. If possible, make this parameter a mandatory parameter.
- A parameter set that contains multiple positional parameters must define unique positions for each parameter. No two positional parameters can specify the same position.
- Only one parameter in a set can declare the `valueFromPipeline` keyword with a value of `true`. Multiple parameters can define the `ValueFromPipelineByPropertyName` keyword with a value of `true`.
- If no parameter set is specified for a parameter, the parameter belongs to all parameter sets.

ⓘ Note

For a cmdlet or function, there is a limit of 32 parameter sets.

Default parameter sets

When multiple parameter sets are defined, you can use the `DefaultParameterSetName` keyword of the **Cmdlet** attribute to specify the default parameter set. PowerShell uses the default parameter set if it can't determine the parameter set to use based on the information provided by the command. For more information about the **Cmdlet** attribute, see [Cmdlet Attribute Declaration](#).

Declaring parameter sets

To create a parameter set, you must specify the `ParameterSetName` keyword when you declare the **Parameter** attribute for every parameter in the parameter set. For parameters that belong to multiple parameter sets, add a **Parameter** attribute for each parameter set. This attribute enables you to define the parameter differently for each parameter set. For example, you can define a parameter as mandatory in one set and optional in another. However, each parameter set must contain one unique parameter. For more information, see [Parameter Attribute Declaration](#).

In the following example, the **UserName** parameter is the unique parameter of the **Test01** parameter set, and the **ComputerName** parameter is the unique parameter of the **Test02** parameter set. The **SharedParam** parameter belongs to both sets and is mandatory for the **Test01** parameter set but optional for the **Test02** parameter set.

C#

```
[Parameter(Position = 0, Mandatory = true, ParameterSetName = "Test01")]
public string UserName
{
    get { return userName; }
    set { userName = value; }
}
private string userName;

[Parameter(Position = 0, Mandatory = true, ParameterSetName = "Test02")]
public string ComputerName
{
    get { return computerName; }
    set { computerName = value; }
}
private string computerName;

[Parameter(Mandatory= true, ParameterSetName = "Test01")]
[Parameter(ParameterSetName = "Test02")]
public string SharedParam
{
    get { return sharedParam; }
    set { sharedParam = value; }
}
private string sharedParam;
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Cmdlet dynamic parameters

Article • 01/31/2024

Cmdlets can define parameters that are available to the user under special conditions, such as when the argument of another parameter is a specific value. These parameters are added at runtime and are referred to as dynamic parameters because they're only added when needed. For example, you can design a cmdlet that adds several parameters only when a specific switch parameter is specified.

ⓘ Note

Providers and PowerShell functions can also define dynamic parameters.

Dynamic parameters in PowerShell cmdlets

PowerShell uses dynamic parameters in several of its provider cmdlets. For example, the `Get-Item` and `Get-ChildItem` cmdlets add a `CodeSigningCert` parameter at runtime when the `Path` parameter specifies the `Certificate` provider path. If the `Path` parameter specifies a path for a different provider, the `CodeSigningCert` parameter isn't available.

The following examples show how the `CodeSigningCert` parameter is added at runtime when `Get-Item` is run.

In this example, the PowerShell runtime has added the parameter and the cmdlet is successful.

PowerShell

```
Get-Item -Path Cert:\CurrentUser -CodeSigningCert
```

Output

```
Location    : CurrentUser
StoreNames  : {SmartCardRoot, UserDS, AuthRoot, CA...}
```

In this example, a `FileSystem` drive is specified and an error is returned. The error message indicates that the `CodeSigningCert` parameter can't be found.

PowerShell

```
Get-Item -Path C:\ -CodeSigningCert
```

Output

```
Get-Item : A parameter cannot be found that matches parameter name
'CodeSigningCert'.
At line:1 char:37
+ Get-Item -Path C:\ -CodeSigningCert <<<
-----
CategoryInfo          : InvalidArgument: (:) [Get-Item],
ParameterBindingException
FullyQualifiedErrorId :
NamedParameterNotFound,Microsoft.PowerShell.Commands.GetItemCommand
```

Support for dynamic parameters

To support dynamic parameters, the following elements must be included in the cmdlet code.

Interface

[System.Management.Automation.IDynamicParameters](#). This interface provides the method that retrieves the dynamic parameters.

For example:

```
public class SendGreetingCommand : Cmdlet, IDynamicParameters
```

Method

[System.Management.Automation.IDynamicParameters.GetDynamicParameters](#). This method retrieves the object that contains the dynamic parameter definitions.

For example:

C#

```
public object GetDynamicParameters()
{
    if (employee)
    {
        context= new SendGreetingCommandDynamicParameters();
        return context;
    }
    return null;
```

```
}
```

```
private SendGreetingCommandDynamicParameters context;
```

Class

A class that defines the dynamic parameters to be added. This class must include a **Parameter** attribute for each parameter and any optional **Alias** and **Validation** attributes that are needed by the cmdlet.

For example:

C#

```
public class SendGreetingCommandDynamicParameters
{
    [Parameter]
    [ValidateSet ("Marketing", "Sales", "Development")]
    public string Department
    {
        get { return department; }
        set { department = value; }
    }
    private string department;
}
```

For a complete example of a cmdlet that supports dynamic parameters, see [How to Declare Dynamic Parameters](#).

See also

- [System.Management.Automation.IDynamicParameters](#)
- [System.Management.Automation.IDynamicParameters.GetDynamicParameters](#)
- [How to Declare Dynamic Parameters](#)
- [Writing a Windows PowerShell Cmdlet](#)

Supporting Wildcard Characters in Cmdlet Parameters

Article • 12/18/2023

Often, you will have to design a cmdlet to run against a group of resources rather than against a single resource. For example, a cmdlet might need to locate all the files in a data store that have the same name or extension. You must provide support for wildcard characters when you design a cmdlet that will be run against a group of resources.

ⓘ Note

Using wildcard characters is sometimes referred to as *globbing*.

Windows PowerShell Cmdlets That Use Wildcards

Many Windows PowerShell cmdlets support wildcard characters for their parameter values. For example, almost every cmdlet that has a `Name` or `Path` parameter supports wildcard characters for these parameters. (Although most cmdlets that have a `Path` parameter also have a `LiteralPath` parameter that does not support wildcard characters.) The following command shows how a wildcard character is used to return all the cmdlets in the current session whose name contains the Get verb.

```
Get-Command get-*
```

Supported Wildcard Characters

Windows PowerShell supports the following wildcard characters.

[] Expand table

Wildcard	Description	Example	Matches	Does not match
*	Matches zero or more characters, starting at the specified position	a*	A, ag, Apple	
?	Matches any character at the specified position	?n	An, in, on ran	

Wildcard	Description	Example	Matches	Does not match
[]	Matches a range of characters	[a-] []ook	book, cook, look	nook, took
[]	Matches the specified characters	[bn]ook	book, nook	cook, look

When you design cmdlets that support wildcard characters, allow for combinations of wildcard characters. For example, the following command uses the `Get-ChildItem` cmdlet to retrieve all the .txt files that are in the C:\Techdocs folder and that begin with the letters "a" through "l."

```
Get-ChildItem C:\techdocs\[a-l]*.txt
```

The previous command uses the range wildcard `[a-1]` to specify that the file name should begin with the characters "a" through "l" and uses the `*` wildcard character as a placeholder for any characters between the first letter of the filename and the `.txt` extension.

The following example uses a range wildcard pattern that excludes the letter "d" but includes all the other letters from "a" through "f."

```
Get-ChildItem C:\techdocs\[a-cef]*.txt
```

Handling Literal Characters in Wildcard Patterns

If the wildcard pattern you specify contains literal characters that should not be interpreted as wildcard characters, use the backtick character (`\``) as an escape character. When you specify literal characters int the PowerShell API, use a single backtick. When you specify literal characters at the PowerShell command prompt, use two backticks.

For example, the following pattern contains two brackets that must be taken literally.

When used in the PowerShell API use:

- "John Smith `[*]`"

When used from the PowerShell command prompt:

- "John Smith ``[*]``"

This pattern matches "John Smith [Marketing]" or "John Smith [Development]". For example:

```
PS> "John Smith [Marketing]" -like "John Smith ``[*``]"
True

PS> "John Smith [Development]" -like "John Smith ``[*``]"
True
```

Cmdlet Output and Wildcard Characters

When cmdlet parameters support wildcard characters, the operation usually generates an array output. Occasionally, it makes no sense to support an array output because the user might use only a single item. For example, the `Set-Location` cmdlet does not support array output because the user sets only a single location. In this instance, the cmdlet still supports wildcard characters, but it forces resolution to a single location.

See Also

[Writing a Windows PowerShell Cmdlet](#)

[WildcardPattern Class](#)

Validating Parameter Input

Article • 09/17/2021

PowerShell can validate the arguments passed to cmdlet parameters in several ways. PowerShell can validate the length, the range, and the pattern of the characters of the argument. It can validate the number of arguments available (the count). These validation rules are defined by validation attributes that are declared with the `Parameter` attribute on public properties of the cmdlet class.

To validate a parameter argument, the PowerShell runtime uses the information provided by the validation attributes to confirm the value of the parameter before the cmdlet is run. If the parameter input is not valid, the user receives an error message. Each validation parameter defines a validation rule that is enforced by PowerShell.

PowerShell enforces the validation rules based on the following attributes.

ValidateCount

Specifies the minimum and maximum number of arguments that a parameter can accept. For more information, see [ValidateCount Attribute Declaration](#).

ValidateLength

Specifies the minimum and maximum number of characters in the parameter argument. For more information, see [ValidateLength Attribute Declaration](#).

ValidatePattern

Specifies a regular expression that validates the parameter argument. For more information, see [ValidatePattern Attribute Declaration](#).

ValidateRange

Specifies the minimum and maximum values of the parameter argument. For more information, see [ValidateRange Attribute Declaration](#).

ValidateScript

Specifies the valid values for the parameter argument. For more information, see [ValidateScript Attribute Declaration](#).

ValidateSet

Specifies the valid values for the parameter argument. For more information, see [ValidateSet Attribute Declaration](#).

See Also

[How to Validate Parameter Input](#)

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Input Filter Parameters

Article • 09/17/2021

A cmdlet can define `Filter`, `Include`, and `Exclude` parameters that filter the set of input objects that the cmdlet affects.

Typically, the set of input objects is specified by an `InputObject`, `Path`, or `Name` parameter. For example, a cmdlet can have a `Path` parameter that accepts multiple paths by using wildcard characters, and each path points to an input object. Used together, the `Filter`, `Include`, and `Exclude` parameters further qualify the paths the cmdlet works on each time it is invoked.

Include and Exclude Parameters

The `Include` and `Exclude` parameters identify the objects that are included or excluded from the set of input objects passed to the cmdlet. Use these parameters when the filter can be expressed in the standard wildcard language. (For more information about wildcard characters, see [Supporting Wildcards in Cmdlet Parameters](#).) The `Include` parameter includes all the objects whose names match the inclusion filter. The `Exclude` parameter excludes all the objects whose names match the filter.

Filter Parameter

The `Filter` parameter specifies a filter that is not expressed in the standard wildcard language. For example, Active Directory Service Interfaces (ADSI) or SQL filters might be passed to the cmdlet through its `Filter` parameter. In the cmdlets provided by Windows PowerShell, these filters are specified by the Windows PowerShell providers that use the cmdlet to access a data store. Each provider typically defines its own filter.

Filtering If No Set of Input Objects Is Specified

If no set of input objects is specified, this typically means to filter against all objects. For more information, see [Get-Process](#).

See Also

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Cmdlet Attributes

Article • 09/17/2021

Windows PowerShell defines several attributes that you can use to add common functionality to your cmdlets without implementing that functionality within your own code. This includes the Cmdlet attribute that identifies a Microsoft .NET Framework class as a cmdlet class, the OutputType attribute that specifies the .NET Framework types returned by the cmdlet, the Parameter attribute that identifies public properties as cmdlet parameters, and more.

In This Section

[Attributes in Cmdlet Code](#) Describes the benefit of using attributes in cmdlet code.

[Attribute Types](#) Describes the different attributes that can decorate a cmdlet class.

[Alias Attribute Declaration](#) Describes how to define aliases for a cmdlet parameter name.

[Cmdlet Attribute Declaration](#) Describes how to define a .NET Framework class as a cmdlet.

[Credential Attribute Declaration](#) Describes how to add support for converting string input into a `System.Management.Automation.PSCredential` object.

[OutputType attribute Declaration](#) Describes how to specify the .NET Framework types returned by the cmdlet.

[Parameter Attribute Declaration](#) Describes how to define the parameters of a cmdlet.

[ValidateCount Attribute Declaration](#) Describes how to define how many arguments are allowed for a parameter.

[ValidateLength Attribute Declaration](#) Describes how to define the length (in characters) of a parameter argument.

[ValidatePattern Attribute Declaration](#) Describes how to define the valid patterns for a parameter argument.

[ValidateRange Attribute Declaration](#) Describes how to define the valid range for a parameter argument.

[ValidateScript Attribute Declaration](#) Describes how to define the possible values for a parameter argument.

[ValidateSet Attribute Declaration](#) Describes how to define the possible values for a parameter argument.

Reference

[Writing a Windows PowerShell Cmdlet](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Attributes in Cmdlet Code

Article • 09/17/2021

To use the common functionality provided by Windows PowerShell, the classes and public properties defined in the cmdlet code are decorated with attributes. For example, the following class definition uses the `Cmdlet` attribute to identify the Microsoft .NET Framework class in which the `Get-Proc` cmdlet is implemented. (This cmdlet is used as an example in this document, and is similar to the `Get-Process` cmdlet provided by Windows PowerShell.)

C#

```
[Cmdlet(VerbsCommon.Get, "Proc")]
public class GetProcCommand : Cmdlet
```

These attributes are considered metadata because their implementation is separate from the implementation of the cmdlet code. When the Windows PowerShell runtime runs the cmdlet, it recognizes the attributes and then performs the appropriate action for each attribute.

Although you might want to implement your own version of the functionality provided by these attributes, a good cmdlet design uses these common functionalities.

For more information about the different attributes that can be declared in your cmdlets, see [Attribute Types](#).

See Also

[Attribute Types](#)

[Writing a Windows PowerShell Cmdlet](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review
issues and pull requests. For



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

more information, see [our contributor guide](#).

Attribute Types

Article • 09/17/2021

Cmdlet attributes can be grouped by functionality. The following sections describe the available attributes and describe what the runtime does when the attribute is invoked.

Cmdlet Attributes

Cmdlet

Identifies a .NET Framework class as a cmdlet. This is the required base attribute. For more information, see [Cmdlet Attribute Declaration](#).

Parameter Attributes

Parameter

Identifies a public property in the cmdlet class as a cmdlet parameter. For more information, see [Parameter Attribute Declaration](#).

Alias

Specifies one or more aliases for a parameter. For more information, see [Alias Attribute Declaration](#).

Argument Validation Attributes

ValidateCount

Specifies the minimum and maximum number of arguments that are allowed for a cmdlet parameter. For more information, see [ValidateCount Attribute Declaration](#).

ValidateLength

Specifies a minimum and maximum number of characters for a cmdlet parameter argument. For more information, see [ValidateLength Attribute Declaration](#).

ValidatePattern

Specifies a regular expression pattern that the cmdlet parameter argument must match. For more information, see [ValidatePattern Attribute Declaration](#).

ValidateRange

Specifies the minimum and maximum values for a cmdlet parameter argument. For more information, see [ValidateRange Attribute Declaration](#).

ValidateSet

Specifies a set of valid values for the cmdlet parameter argument. For more information, see [ValidateSet Attribute Declaration](#).

See Also

[Windows PowerShell SDK](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Alias Attribute Declaration

Article • 03/24/2025

The **Alias** attribute allows the user to specify different names for a cmdlet or a cmdlet parameter. Aliases can be used to provide shortcuts for a parameter name, or they can provide different names that are appropriate for different scenarios.

Syntax

C#

```
[Alias(aliasNames)]
```

Parameters

`aliasNames` (String[]) Required. Specifies a set of comma-separated alias names for the cmdlet parameter.

Remarks

The **Alias** attribute is defined by the [System.Management.Automation.AliasAttribute](#) class.

Cmdlet aliases

- The **Alias** attribute is used with the cmdlet declaration. For more information about how to declare these attributes, see [Cmdlet Aliases](#).
- Each parameter alias name must be unique. Windows PowerShell does not check for duplicate alias names.

Parameter aliases

- The **Alias** attribute is used with the **Parameter** attribute when you specify a cmdlet parameter. For more information about how to declare these attributes, see [How to Declare Cmdlet Parameters](#).
- Each parameter alias name must be unique within a cmdlet. Windows PowerShell does not check for duplicate alias names.
- The **Alias** attribute is used once for each parameter in a cmdlet.

See Also

[Cmdlet Aliases](#)

[Parameter Aliases](#)

[Writing a Windows PowerShell Cmdlet](#)

Cmdlet Attribute Declaration

Article • 09/17/2021

The `Cmdlet` attribute identifies a Microsoft .NET Framework class as a cmdlet and specifies the verb and noun used to invoke the cmdlet.

Syntax

C#

```
[Cmdlet("verbName", "nounName")]
[Cmdlet("verbName", "nounName", Named Parameters...)]
```

Parameters

`VerbName` ([System.String](#)) Required. Specifies the cmdlet verb. This verb specifies the action taken by the cmdlet. For more information about approved cmdlet verbs, see [Cmdlet Verb Names](#) and [Required Development Guidelines](#).

`NounName` ([System.String](#)) Required. Specifies the cmdlet noun. This noun specifies the resource that the cmdlet acts upon. For more information about cmdlet nouns, see [Cmdlet Declaration](#) and [Strongly Encouraged Development Guidelines](#).

`SupportsShouldProcess` ([System.Boolean](#)) Optional named parameter. `True` indicates that the cmdlet supports calls to the [System.Management.Automation.Cmdlet.ShouldProcess](#) method, which provides the cmdlet with a way to prompt the user before an action that changes the system is performed. `False`, the default value, indicates that the cmdlet does not support calls to the [System.Management.Automation.Cmdlet.ShouldProcess](#) method. For more information about confirmation requests, see [Requesting Confirmation](#).

`ConfirmImpact` ([System.Management.Automation.ConfirmImpact](#)) Optional named parameter. Specifies when the action of the cmdlet should be confirmed by a call to the [System.Management.Automation.Cmdlet.ShouldProcess](#) method. [System.Management.Automation.Cmdlet.ShouldProcess](#) will only be called when the `ConfirmImpact` value of the cmdlet (by default, Medium) is equal to or greater than the value of the `$ConfirmPreference` variable. This parameter should be specified only when the `SupportsShouldProcess` parameter is specified.

`DefaultParameterSetName` ([System.String](#)) Optional named parameter. Specifies the default parameter set that the Windows PowerShell runtime attempts to use when it cannot determine which parameter set to use. Notice that this situation can be eliminated by making the unique parameter of each parameter set a mandatory parameter.

There is one case where Windows PowerShell cannot use the default parameter set even if a default parameter set name is specified. The Windows PowerShell runtime cannot distinguish between parameter sets based solely on object type. For example, if you have one parameter set that takes a string as the file path, and another set that takes a [FileInfo](#) object directly, Windows PowerShell cannot determine which parameter set to use based on the values passed to the cmdlet, nor does it use the default parameter set. In this case, even if you specify a default parameter set name, Windows PowerShell throws an ambiguous parameter set error message.

`SupportsTransactions` ([System.Boolean](#)) Optional named parameter. `True` indicates that the cmdlet can be used within a transaction. When `True` is specified, the Windows PowerShell runtime adds the `UseTransaction` parameter to the parameter list of the cmdlet. `False`, the default value, indicates that the cmdlet cannot be used within a transaction.

Remarks

- Together, the verb and noun are used to identify your registered cmdlet and to invoke your cmdlet within a script.
- When the cmdlet is invoked from the Windows PowerShell console, the command resembles the following command:

VerbName-NounName

- All cmdlets that change resources outside of Windows PowerShell should include the `SupportsShouldProcess` keyword when the `Cmdlet` attribute is declared, which allows the cmdlet to call the [System.Management.Automation.Cmdlet.ShouldProcess](#) method before the cmdlet performs its action. If the [System.Management.Automation.Cmdlet.ShouldProcess](#) call returns `false`, the action should not be taken. For more information about the confirmation requests generated by the [System.Management.Automation.Cmdlet.ShouldProcess](#) call, see [Requesting Confirmation](#).

The `Confirm` and `WhatIf` cmdlet parameters are available only for cmdlets that support `System.Management.Automation.Cmdlet.ShouldProcess` calls.

Example

The following class definition uses the `Cmdlet` attribute to identify the .NET Framework class for a **Get-Proc** cmdlet that retrieves information about the processes running on the local computer.

```
C#  
  
[Cmdlet(VerbsCommon.Get, "Proc")]
public class GetProcCommand : Cmdlet
```

For more information about the **Get-Proc** cmdlet, see [GetProc Tutorial](#).

See Also

[Writing a Windows PowerShell Cmdlet](#)

Credential Attribute Declaration

Article • 09/17/2021

The Credential attribute is an optional attribute that can be used with credential parameters of type [System.Management.Automation.PSCredential](#) so that a string can also be passed as an argument to the parameter. When this attribute is added to a parameter declaration, Windows PowerShell converts the string input into a [System.Management.Automation.PSCredential](#) object. For example, the [Get-Credential](#) cmdlet uses this attribute to have Windows PowerShell generate the [System.Management.Automation.PSCredential](#) object that is returned by the cmdlet.

Syntax

C#

```
[Credential]
```

Remarks

- Typically this attribute is used by parameters of type [System.Management.Automation.PSCredential](#) so that a string can also be passed as an argument to the parameter. When a [System.Management.Automation.PSCredential](#) object is passed to the parameter, Windows PowerShell does nothing.
- When creating the [System.Management.Automation.PSCredential](#) object, Windows PowerShell uses the current Host to display the appropriate prompts to the user. For example, the default Host displays a prompt for a user name and password when this attribute is used. However, if a custom host is being used that defines a different prompt then that prompt would be displayed.
- This attribute is used with the Parameter attribute. For more information about that attribute, see [Parameter Attribute Declaration](#).
- The credential attribute is defined by the [System.Management.Automation.CredentialAttribute](#) class.

See Also

[Parameter Aliases](#)

[Parameter Attribute Declaration](#)

[Writing a Windows PowerShell Cmdlet](#)

OutputType Attribute Declaration

Article • 09/17/2021

The `OutputType` attribute identifies the .NET Framework types returned by a cmdlet, function, or script.

Syntax

C#

```
[OutputType(params string[] type)]
[OutputType(params Type[] type)]
[OutputType(params string[] type, Named Parameters...)]
[OutputType(params Type[] type, Named Parameters...)]
```

Parameters

Type (`string[]` or `Type[]`) Required. Specifies the types returned by the cmdlet function, or script.

ParameterSetName (`string[]`) Optional. Specifies the parameter sets that return the types specified in the `type` parameter.

providerCmdlet Optional. Specifies the provider cmdlet that returns the types specified in the `type` parameter.

See Also

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Parameter Attribute Declaration

Article • 01/06/2025

The Parameter attribute identifies a public property of the cmdlet class as a cmdlet parameter.

Syntax

C#

```
[Parameter()]
[Parameter(Named Parameters...)]
```

Parameters

`Mandatory` (`System.Boolean`) Optional named parameter. `True` indicates the cmdlet parameter is required. If a required parameter is not provided when the cmdlet is invoked, Windows PowerShell prompts the user for a parameter value. The default is `false`.

`ParameterSetName` (`System.String`) Optional named parameter. Specifies the parameter set that this cmdlet parameter belongs to. If no parameter set is specified, the parameter belongs to all parameter sets.

`Position` (`System.Int32`) Optional named parameter. Specifies the position of the parameter within a Windows PowerShell command.

`ValueFromPipeline` (`System.Boolean`) Optional named parameter. `True` indicates that the cmdlet parameter takes its value from a pipeline object. Specify this keyword if the cmdlet accesses the complete object, not just a property of the object. The default is `false`.

`ValueFromPipelineByPropertyName` (`System.Boolean`) Optional named parameter. `True` indicates that the cmdlet parameter takes its value from a property of a pipeline object that has either the same name or the same alias as this parameter. For example, if the cmdlet has a `Name` parameter and the pipeline object also has a `Name` property, the value of the `Name` property is assigned to the `Name` parameter of the cmdlet. The default is `false`.

`ValueFromRemainingArguments` ([System.Boolean](#)) Optional named parameter. `True` indicates that the cmdlet parameter accepts all remaining arguments that are passed to the cmdlet. The default is `false`.

`HelpMessage` ([System.String](#)) Optional named parameter. Specifies a short description of the parameter. Windows PowerShell displays this message when a cmdlet is run and a mandatory parameter is not specified.

`HelpMessageBaseName` ([System.String](#)) Optional named parameter. Specifies the location where resource identifiers reside. For example, this parameter could specify a resource assembly that contains Help messages that you want to localize.

`HelpMessageResourceId` ([System.String](#)) Optional named parameter. Specifies the resource identifier for a Help message.

`DontShow` ([System.Boolean](#)) Optional named parameter. `True` indicates that the parameter is hidden from the user for tab expansion and IntelliSense. The default is `false`.

Remarks

- For more information about how to declare this attribute, see [How to Declare Cmdlet Parameters](#).
- A cmdlet can have any number of parameters. However, for a better user experience, limit the number of parameters.
- Parameters must be declared on public non-static fields or properties. Parameters should be declared on properties. The property must have a public set accessor, and if the `ValueFromPipeline` or `ValueFromPipelineByPropertyName` keyword is specified, the property must have a public get accessor.
- When you specify positional parameters, limit the number of positional parameters in a parameter set to less than five. And, positional parameters do not have to be contiguous. Positions 5, 100, and 250 work the same as positions 0, 1, and 2.
- When the `Position` keyword is not specified, the cmdlet parameter must be referenced by its name.
- When you use parameter sets, note the following:
 - Each parameter set must have at least one unique parameter. Good cmdlet design indicates this unique parameter should also be mandatory if possible. If

your cmdlet is designed to be run without parameters, the unique parameter cannot be mandatory.

- No parameter set should contain more than one positional parameter with the same position.
- Only one parameter in a parameter set should declare `ValueFromPipeline = true`.
- Multiple parameters can define `ValueFromPipelineByPropertyName = true`.
- For more information about the guidelines for parameter names, see [Cmdlet Parameter Names](#).
- The parameter attribute is defined by the [System.Management.Automation.ParameterAttribute](#) class.
- The `DontShow` parameter has the following side effects:
 - Affects all parameter sets for the associated parameter, even if there's a parameter set in which `DontShow` is unused.
 - Hides common parameters from tab completion and IntelliSense. `DontShow` doesn't hide the optional common parameters: `WhatIf`, `Confirm`, or `UseTransaction`.

See Also

- [System.Management.Automation.ParameterAttribute](#)
- [Cmdlet Parameter Names](#)
- [Writing a Windows PowerShell Cmdlet](#)

ValidateCount Attribute Declaration

Article • 09/17/2021

The ValidateCount attribute specifies the minimum and maximum number of arguments allowed for a cmdlet parameter.

Syntax

C#

```
[ValidateCount(int minLength, int maxLength)]
```

Parameters

`MinLength` ([System.Int32](#)) Required. Specifies the minimum number of arguments.

`MaxLength` ([System.Int32](#)) Required. Specifies the maximum number of arguments.

Remarks

- For more information about how to declare this attribute, see [How to Validate an Argument Count](#).
- When this attribute is not invoked, the corresponding cmdlet parameter can have any number of arguments.
- The Windows PowerShell runtime throws an error under the following conditions:
 - The `MinLength` and `MaxLength` attribute parameters are not of type [System.Int32](#).
 - The value of the `MaxLength` attribute parameter is less than the value of the `MinLength` attribute parameter.
- The ValidateCount attribute is defined by the [System.Management.Automation.ValidateCountAttribute](#) class.

See Also

[System.Management.Automation.ValidateCountAttribute](#)

[How to Validate an Argument Count](#)

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ValidateLength Attribute Declaration

Article • 09/17/2021

The ValidateLength attribute specifies the minimum and maximum number of characters for a cmdlet parameter argument. This attribute can also be used by Windows PowerShell functions.

Syntax

C#

```
[ValidateLength(int minLength, int maxLength)]
```

Parameters

`MinLength` ([System.Int32](#)) Required. Specifies the minimum number of characters allowed.

`MaxLength` ([System.Int32](#)) Required. Specifies the maximum number of characters allowed.

Remarks

- For more information about how to declare this attribute, see [How to Declare Input Validation Rules](#).
- When this attribute is not used, the corresponding parameter argument can be of any length.
- The Windows PowerShell runtime throws an error under the following conditions:
 - When the value of the `MaxLength` attribute parameter is less than the value of the `MinLength` attribute parameter.
 - When the `MaxLength` attribute parameter is set to 0.
 - When the argument is not a string.
- The ValidateLength attribute is defined by the [System.Management.Automation.ValidateLengthAttribute](#) class.

See Also

[System.Management.Automation.ValidateLengthAttribute](#)

[Writing a Windows PowerShell Cmdlet](#)

ValidatePattern Attribute Declaration

Article • 03/24/2025

The `ValidatePattern` attribute specifies a regular expression pattern that validates the argument of a cmdlet parameter. This attribute can also be used by Windows PowerShell functions.

When `ValidatePattern` is invoked within a cmdlet, the Windows PowerShell runtime converts the argument of the cmdlet parameter to a string and then compares that string to the pattern supplied by the `ValidatePattern` attribute. The cmdlet is run only if the converted string representation of the argument and the supplied pattern match. If they do not match, an error is thrown by the Windows PowerShell runtime.

Syntax

C#

```
[ValidatePattern(string regexString)]
[ValidatePattern(string regexString, Named Parameters)]
```

Parameters

`RegexString` ([System.String](#)) Required. Specifies a regular expression that validates the parameter argument.

`Options` ([System.Text.RegularExpressions.RegexOptions](#)) Optional named parameter. Specifies a bitwise combination of [System.Text.RegularExpressions.RegexOptions](#) flags that specify regular expression options.

Remarks

- This attribute can be used only once per parameter.
- You can use the `Option` parameter of the attribute to further define the pattern. For example, you can make the pattern case sensitive.
- If this attribute is applied to a collection, each element in the collection must match the pattern.

- The ValidatePattern attribute is defined by the [System.Management.Automation.ValidatePatternAttribute](#) class.

See Also

[System.Management.Automation.ValidatePatternAttribute](#)

[Writing a Windows PowerShell Cmdlet](#)

ValidateRange Attribute Declaration

Article • 09/17/2021

The ValidateRange attribute specifies the minimum and maximum values (the range) for the cmdlet parameter argument. This attribute can also be used by Windows PowerShell functions.

Syntax

C#

```
[ValidateRange(object minRange, object maxRange)]
```

Parameters

`MinRange` ([System.Object](#)) Required. Specifies the minimum value allowed.

`MaxRange` ([System.Object](#)) Required. Specifies the maximum value allowed.

Remarks

- The Windows PowerShell runtime throws a construction error when the value of the `MinRange` parameter is greater than the value of the `MaxRange` parameter.
- The Windows PowerShell runtime throws a validation error under the following conditions:
 - When the value of the argument is less than the `MinRange` limit or greater than the `MaxRange` limit.
 - When the argument is not of the same type as the `MinRange` and the `MaxRange` parameters.
- The ValidateRange attribute is defined by the [System.Management.Automation.ValidateRangeAttribute](#) class.

See Also

[System.Management.Automation.ValidateRangeAttribute](#)

Writing a Windows PowerShell Cmdlet

ValidateScript Attribute Declaration

Article • 10/03/2022

The `ValidateScript` attribute specifies a script that's used to validate a parameter or variable value. PowerShell pipes the value to the script, and generates an error if the script returns `$false` or if the script throws an exception.

When you use the `ValidateScript` attribute, the value that's being validated is mapped to the `$_` variable. You can use the `$_` variable to refer to the value in the script.

Syntax

C#

```
[ValidateScriptAttribute( ScriptBlock scriptBlock )]  
[ValidateScriptAttribute( ScriptBlock scriptBlock, Named Parameters )]
```

Parameters

- `scriptBlock` - ([System.Management.Automation.ScriptBlock](#)) Required. The script block used to validate the input.
- `ErrorMessage` - Optional named parameter - The item being validated and the validating scriptblock are passed as the first and second formatting arguments.

ⓘ Note

The `ErrorMessage` argument was added in PowerShell 6.

Remarks

- This attribute can be used only once per parameter.
- If this attribute is applied to a collection, each element in the collection must match the pattern.
- The `ValidateScript` attribute is defined by the [System.Management.Automation.ValidateScriptAttribute](#) class.

See Also

Writing a Windows PowerShell Cmdlet

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ValidateSet Attribute Declaration

Article • 03/24/2025

The `ValidateSetAttribute` attribute specifies a set of possible values for a cmdlet parameter argument. This attribute can also be used by Windows PowerShell functions.

When this attribute is specified, the Windows PowerShell runtime determines whether the supplied argument for the cmdlet parameter matches an element in the supplied element set. The cmdlet is run only if the parameter argument matches an element in the set. If no match is found, an error is thrown by the Windows PowerShell runtime.

Syntax

C#

```
[ValidateSetAttribute(params string[] validValues)]
[ValidateSetAttribute(params string[] validValues, Named Parameters)]
```

Parameters

`ValidValues` ([System.String](#)) Required. Specifies the valid parameter element values. The following sample shows how to specify one element or multiple elements.

C#

```
[ValidateSetAttribute("Steve")]
[ValidateSetAttribute("Steve", "Mary")]
```

`IgnoreCase` ([System.Boolean](#)) Optional named parameter. The default value of `true` indicates that case is ignored. A value of `false` makes the cmdlet case-sensitive.

Remarks

- This attribute can be used only once per parameter.
- If the parameter value is an array, every element of the array must match an element of the attribute set.
- The `ValidateSetAttribute` attribute is defined by the [System.Management.Automation.ValidateSetAttribute](#) class.

See Also

[System.Management.Automation.ValidateSetAttribute](#)

[Writing a Windows PowerShell Cmdlet](#)

Cmdlet Aliases

Article • 09/17/2021

You can use cmdlet aliases to improve the cmdlet user experience. You can add aliases to frequently used cmdlets to reduce typing and to make it easier to complete tasks quickly. You can include built-in aliases in your cmdlets, or users can define their own custom aliases.

For example, the [Get-Command](#) cmdlet has a built-in `gcm` alias. You can also use aliases to add command names from other languages so that users do not have to learn new commands.

Alias Guidelines

Follow these guidelines when you create built-in aliases for your cmdlets:

- Before you assign aliases, start Windows PowerShell, and then run the [Get-Alias](#) cmdlet to see the aliases that are already used.
- Include an alias prefix that references the verb of the cmdlet name and an alias suffix that references the noun of the cmdlet name. For example, the alias for the `Import-Module` cmdlet is `ipmo`. For a list of all the verbs and their aliases, see [Cmdlet Verbs](#).
- For cmdlets that have the same verb, include the same alias prefix. For example, the aliases for all the Windows PowerShell cmdlets that have the "Get" verb in their name use the "g" prefix.
- For cmdlets that have the same noun, include the same alias suffix. For example, the aliases for all the Windows PowerShell cmdlets that have the "Session" noun in their name use the "sn" suffix.
- For cmdlets that are equivalent to commands in other languages, use the name of the command.
- In general, make aliases as short as possible. Make sure the alias has at least one distinct character for the verb and one distinct character for the noun. Add more characters as needed to make the alias unique.
- For cmdlet written in C# (or any other compiled .NET language), the alias can be defined using the [Alias attribute](#). For example:

C#

```
[Cmdlet("Get", "SomeObject")]
[Alias("gso")]
public class GetSomeObjectCommand : Cmdlet
```

See Also

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Cmdlet Output

Article • 09/17/2021

This section discusses the types of cmdlet output and the methods that cmdlets can call to generate output such as error messages and objects. This section also describes how to define the .NET Framework types that are returned by your cmdlets and how those objects are displayed.

In This Section

[Types of Cmdlet Output](#) Describes the types and output that cmdlets can generate and the methods that cmdlets call to generate the output.

[Cmdlet Error Reporting](#) Discusses cmdlet error reporting, a subset of cmdlet output.

[Extending Output Objects](#) Discusses how to use the types files (`.ps1xml`) to extend the .NET Framework objects that are returned by cmdlets, functions, and scripts.

[PowerShell Formatting Files](#) Describes the formatting files (`.format.ps1xml`) files that define the default display for a specific set of .NET Framework objects in Windows PowerShell.

[Custom Formatting Files](#) Describes how to create your own custom formatting files to overwrite the default display formats or to define the display of objects returned by your own commands.

See Also

[Writing a Windows PowerShell Cmdlet](#)

Types of cmdlet output

Article • 09/17/2021

PowerShell provides several methods that can be called by cmdlets to generate output. These methods use a specific operation to write their output to a specific data stream, such as the success data stream or the error data stream. This article describes the types of output and the methods used to generate them.

Types of output

Success output

Cmdlets can report success by returning an object that can be processed by the next command in the pipeline. After the cmdlet has successfully performed its action, the cmdlet calls the [System.Management.Automation.Cmdlet.WriteObject](#) method. We recommend that you call this method instead of the [System.Console.WriteLine](#) or [System.Management.Automation.Host.PSHostUserInterface.WriteLine](#) methods.

You can provide a **PassThru** switch parameter for cmdlets that do not typically return objects. When the **PassThru** switch parameter is specified at the command line, the cmdlet is asked to return an object. For an example of a cmdlet that has a **PassThru** parameter, see [Add-History](#).

Error output

Cmdlets can report errors. When a terminating error occurs, the cmdlet throws an exception. When a non-terminating error occurs, the cmdlet calls the [System.Management.Automation.Provider.CmdletProvider.WriteError](#) method to send an error record to the error data stream. For more information about error reporting, see [Error Reporting Concepts](#).

Verbose output

Cmdlets can provide useful information to you while the cmdlet is correctly processing records by calling the [System.Management.Automation.Cmdlet.WriteVerbose](#) method. The method generates verbose messages that indicate how the action is proceeding.

By default, verbose messages are not displayed. You can specify the **Verbose** parameter when the cmdlet is run to display these messages. **Verbose** is a common parameter that

is available to all cmdlets.

Progress output

Cmdlets can provide progress information to you when the cmdlet is performing tasks that take a long time to complete, such as copying a directory recursively. To display progress information the cmdlet calls the [System.Management.Automation.Cmdlet.WriteProgress](#) method.

Debug output

Cmdlets can provide debug messages that are helpful when troubleshooting the cmdlet code. To display debug information the cmdlet calls the [System.Management.Automation.Cmdlet.WriteDebug](#) method.

By default, debug messages are not displayed. You can specify the **Debug** parameter when the cmdlet is run to display these messages. **Debug** is a common parameter that is available to all cmdlets.

Warning output

Cmdlets can display warning messages by calling the [System.Management.Automation.Cmdlet.WriteWarning](#) method.

By default, warning messages are displayed. However, you can configure warning messages by using the `$WarningPreference` variable or by using the **Verbose** and **Debug** parameters when the cmdlet is called.

Displaying output

For all write-method calls, the content display is determined by specific runtime variables. The exception is the [System.Management.Automation.Cmdlet.WriteObject](#) method. By using these variables, you can make the appropriate write call at the correct place in your code and not worry about when or if the output should be displayed.

Accessing the output functionality of a host application

You can also design a cmdlet to directly access the output functionality of a host application through the PowerShell runtime. Using the host APIs provided by PowerShell

instead of `System.Console` or `System.Windows.Forms` ensures that your cmdlet will work with a variety of hosts. For example: the `powershell.exe` console host, the `powershell_ise.exe` graphical host, the PowerShell remoting host, and third-party hosts.

See also

[Error Reporting Concepts](#)

[Cmdlet Overview](#)

[Writing a Windows PowerShell Cmdlet](#)

Cmdlet error reporting

Article • 09/15/2023

Cmdlets should report errors differently depending on whether the errors are terminating errors or non-terminating errors. Terminating errors are errors that cause the pipeline to be terminated immediately, or errors that occur when there's no reason to continue processing. Non-terminating errors are those errors that report a current error condition, but the cmdlet can continue to process input objects. With non-terminating errors, the user is typically notified of the problem, but the cmdlet continues to process the next input object.

Unless specified otherwise, all classes and methods mentioned in this document come from the [System.Management.Automation](#) namespace.

Terminating and non-terminating errors

The following guidelines can be used to determine if an error condition is a terminating error or a non-terminating error.

- Does the error condition prevent your cmdlet from successfully processing any further input objects? If so, this is a terminating error.
- Is the error condition related to a specific input object or a subset of input objects? If so, this is a non-terminating error.
- Does the cmdlet accept multiple input objects, such that processing may succeed on another input object? If so, this is a non-terminating error.
- Cmdlets that can accept multiple input objects should decide between what are terminating and non-terminating errors, even when a particular situation applies to only a single input object.
- Cmdlets can receive any number of input objects and send any number of success or error objects before throwing a terminating exception. There's no relationship between the number of input objects received and the number of success and error objects sent.
- Cmdlets that can accept only 0-1 input objects and generate only 0-1 output objects should treat errors as terminating errors and generate terminating exceptions.

Reporting non-terminating errors

The reporting of a non-terminating error should always be done within the cmdlet's implementation of the following methods:

- [Cmdlet.BeginProcessing](#)
- [Cmdlet.ProcessRecord](#)
- [Cmdlet.EndProcessing](#)

These types of errors are reported by calling the [Cmdlet.WriteError](#) method that in turn sends an error record to the error stream.

Reporting terminating errors

Terminating errors are reported by throwing exceptions or by calling the [Cmdlet.ThrowTerminatingError](#) method. Be aware that cmdlets can also catch and rethrow exceptions such as [OutOfMemory](#), however, they aren't required to rethrow exceptions as the PowerShell runtime will catch them as well.

You can also define your own exceptions for issues specific to your situation, or add additional information to an existing exception using its error record.

Error records

PowerShell describes a non-terminating error condition with [ErrorRecord](#) objects. Each object provides error category information, an optional target object, and details about the error condition.

Error identifiers

The error identifier is a simple string that identifies the error condition within the cmdlet. PowerShell combines this identifier with a cmdlet identifier to create a fully qualified error identifier that can be used later when filtering error streams or logging errors, when responding to specific errors, or with other user-specific activities.

The following guidelines should be followed when specifying error identifiers:

- Assign different, highly specific, error identifiers to different code paths. Each code path that calls [Cmdlet.WriteError](#) or [Cmdlet.ThrowTerminatingError](#) should have its own error identifier.

- Error identifiers should be unique to Common Language Runtime (CLR) exception types for both terminating and non-terminating errors.
- Don't change the semantics of an error identifier between versions of your cmdlet or PowerShell provider. After the semantics of an error identifier is established, it should remain constant throughout the lifecycle of your cmdlet.
- For terminating errors, use a unique error identifier for a particular CLR exception type. If the exception type changes, use a new error identifier.
- For non-terminating errors, use a specific error identifier for a specific input object.
- Choose text for the identifier that tersely corresponds to the error being reported. Don't use white space or punctuation.
- Don't generate error identifiers that aren't reproducible. For example, don't generate identifiers that include a process identifier. Error identifiers are useful only when they correspond to identifiers that are seen by other users who are experiencing the same problem.

Error categories

Error categories are used to group errors for the user. PowerShell defines these categories and cmdlets and PowerShell providers must choose between them when generating the error record.

For a description of the error categories that are available, see the [ErrorCategory](#) enumeration. In general, you should avoid using **NoError**, **UndefinedError**, and **GenericError** whenever possible.

Users can view errors based on category when they set `$ErrorView` to **CategoryView**.

See also

- [Cmdlet Overview](#)
- [Types of Cmdlet Output](#)
- [Windows PowerShell Reference](#)



Collaborate with us on



PowerShell feedback

GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Extending Output Objects

Article • 02/24/2025

You can extend the .NET Framework objects that are returned by cmdlets, functions, and scripts by using types files (`.ps1xml`). Types files are XML-based files that let you add properties and methods to existing objects. For example, Windows PowerShell provides the `Types.ps1xml` file, which adds elements to several existing .NET Framework objects. The `Types.ps1xml` file is located in the Windows PowerShell installation directory (`$PSHOME`). You can create your own types file to further extend those objects or to extend other objects. When you extend an object by using a types file, any instance of the object is extended with the new elements.

Extending the `System.Array` Object

The following example shows how Windows PowerShell extends the `System.Array` object in the `Types.ps1xml` file. By default, `System.Array` objects have a `Length` property that lists the number of objects in the array. However, because the name "length" does not clearly describe the property, Windows PowerShell adds the `Count` alias property, which displays the same value as the `Length` property. The following XML adds the `Count` property to the `System.Array` type.

```
XML

<Type>
  <Name>System.Array</Name>
  <Members>
    <AliasProperty>
      <Name>Count</Name>
      <ReferencedMemberName>Length</ReferencedMemberName>
    </AliasProperty>
  </Members>
</Type>
```

To see this new alias property, use a `Get-Member` command on any array, as shown in the following example.

```
PowerShell

Get-Member -InputObject (1,2,3,4)
```

The command returns the following results.

Output

Name	MemberType	Definition
Count	AliasProperty	Count = Length
Address	Method	System.Object& Address(Int32)
Clone	Method	System.Object Clone()
CopyTo	Method	System.Void CopyTo(Array array, Int32 index):
Equals	Method	System.Boolean Equals(Object obj)
Get	Method	System.Object Get(Int32)
...		
Length	Property	System.Int32 Length {get;}

You can use either the `Count` property or the `Length` property to determine how many objects are in an array. For example:

PowerShell

```
PS> (1, 2, 3, 4).Count
```

Output

```
4
```

PowerShell

```
PS> (1, 2, 3, 4).Length
```

Output

```
4
```

Custom Types Files

To create a custom types file, start by copying an existing types file. The new file can have any name, but it must have a `.ps1xml` file name extension. When you copy the file, you can place the new file in any directory that is accessible to Windows PowerShell, but it is useful to place the files in the Windows PowerShell installation directory (`$PSHOME`) or in a subdirectory of the installation directory.

To add your own extended types to the file, add a `types` element for each object that you want to extend. The following topics provide examples.

- For more information about adding properties and property sets, see [Extended Properties](#)
- For more information about adding methods, see [Extended Methods](#).
- For more information about adding member sets, see [Extended Member Sets](#).

After you define your own extended types, use one of the following methods to make your extended objects available:

- To make your extended types file available to the current session, use the [Update-TypeData](#) cmdlet to add the new file. If you want your types to take precedence over the types that are defined in other types files (including the Types.ps1xml file), use the `PrependData` parameter of the [Update-TypeData](#) cmdlet.
- To make your extended types file available to all future sessions, add the types file to a module, export the current session, or add the [Update-TypeData](#) command to your Windows PowerShell profile.

Signing Types Files

Types files should be digitally signed to prevent tampering because the XML can include script blocks. For more information about adding digital signatures, see [about_Signing](#)

See Also

[Defining Default Properties for Objects](#)

[Defining Default Methods for Objects](#)

[Defining Default Member Sets for Objects](#)

[Writing a Windows PowerShell Cmdlet](#)

Extending Properties for Objects

Article • 09/17/2021

When you extend .NET Framework objects, you can add alias properties, code properties, note properties, script properties, and property sets to the objects. The XML that defines these properties is described in the following sections.

ⓘ Note

The examples in the following sections are from the default `Types.ps1xml` types file in the PowerShell installation directory (`$PSHOME`). For more information, see [About Types.ps1xml](#).

Alias properties

An alias property defines a new name for an existing property.

In the following example, the `Count` property is added to the `System.Array` type. The `AliasProperty` element defines the extended property as an alias property. The `Name` element specifies the new name. And, the `ReferencedMemberName` element specifies the existing property that is referenced by the alias. You can also add the `AliasProperty` element to the members of the `MemberSets` element.

XML

```
<Type>
  <Name>System.Array</Name>
  <Members>
    <AliasProperty>
      <Name>Count</Name>
      <ReferencedMemberName>Length</ReferencedMemberName>
    </AliasProperty>
  </Members>
</Type>
```

Code properties

A code property references a static property of a .NET Framework object.

In the following example, the `Mode` property is added to the `System.IO.DirectoryInfo` type. The `CodeProperty` element defines the extended property as a code property. The

Name element specifies the name of the extended property. And, the **GetCodeReference** element defines the static method that is referenced by the extended property. You can also add the **CodeProperty** element to the members of the **MemberSets** element.

XML

```
<Type>
  <Name>System.IO.DirectoryInfo</Name>
  <Members>
    <CodeProperty>
      <Name>Mode</Name>
      <GetCodeReference>

    <Type>
      <Name>Microsoft.PowerShell.Commands.FileSystemProvider</Name>
      <Method>
        <Name>Mode</Name>
        <GetCodeReference>
      </Method>
    </Type>
  </Members>
</Type>
```

Note properties

A note property defines a property that has a static value.

In the following example, the **Status** property, whose value is always **Success**, is added to the **System.IO.DirectoryInfo** type. The **NoteProperty** element defines the extended property as a note property. The **Name** element specifies the name of the extended property. The **Value** element specifies the static value of the extended property. The **NoteProperty** element can also be added to the members of the **MemberSets** element.

XML

```
<Type>
  <Name>System.IO.DirectoryInfo</Name>
  <Members>
    <NoteProperty>
      <Name>Status</Name>
      <Value>Success</Value>
    </NoteProperty>
  </Members>
</Type>
```

Script properties

A script property defines a property whose value is the output of a script.

In the following example, the **VersionInfo** property is added to the **System.IO.FileInfo** type. The **ScriptProperty** element defines the extended property as a script property. The **Name** element specifies the name of the extended property. And, the **GetScriptBlock** element specifies the script that generates the property value. You can also add the **ScriptProperty** element to the members of the **MemberSets** element.

XML

```
<Type>
  <Name>System.IO.FileInfo</Name>
  <Members>
    <ScriptProperty>
      <Name>VersionInfo</Name>
      <GetScriptBlock>
        [System.Diagnostics.FileVersionInfo]::GetVersionInfo($this.FullName)
      </GetScriptBlock>
    </ScriptProperty>
  </Members>
</Type>
```

Property sets

A property set defines a group of extended properties that can be referenced by the name of the set. For example, the **Format-Table** **Property** parameter can specify a specific property set to be displayed. When a property set is specified, only those properties that belong to the set are displayed.

There's no restriction on the number of property sets that can be defined for an object. However, the property sets used to define the default display properties of an object must be specified within the **PSStandardMembers** member set. In the **Types.ps1xml** types file, the default property set names include **DefaultDisplayProperty**, **DefaultDisplayPropertySet**, and **DefaultKeyPropertySet**. Any additional property sets that you add to the **PSStandardMembers** member set are ignored.

In the following example, the **DefaultDisplayPropertySet** property set is added to the **PSStandardMembers** member set of the **System.ServiceProcess.ServiceController** type. The **PropertySet** element defines the group of properties. The **Name** element specifies the name of the property set. And, the **ReferencedProperties** element specifies the properties of the set. You can also add the **PropertySet** element to the members of the **Type** element.

XML

```
<Type>
  <Name>System.ServiceProcess.ServiceController</Name>
  <Members>
    <MemberSet>
      <Name>PSStandardMembers</Name>
      <Members>
        <PropertySet>
          <Name>DefaultDisplayPropertySet</Name>
          <ReferencedProperties>
            <Name>Status</Name>
            <Name>Name</Name>
            <Name>DisplayName</Name>
          </ReferencedProperties>
        </PropertySet>
      </Members>
    </MemberSet>
  </Members>
</Type>
```

See also

[About Types.ps1xml](#)

[System.Management.Automation](#)

[Writing a Windows PowerShell Cmdlet](#)

Defining Default Methods for Objects

Article • 09/17/2021

When you extend .NET Framework objects, you can add code methods and script methods to the objects. The XML that is used to define these methods is described in the following sections.

ⓘ Note

The examples in the following sections are from the `Types.ps1xml` types file in the Windows PowerShell installation directory (`$PSHOME`). For more information, see [About Types.ps1xml](#).

Code methods

A code method references a static method of a .NET Framework object.

In the following example, the `ToString` method is added to the `System.Xml.XmlNode` type. The `PSCodeMethod` element defines the extended method as a code method. The `Name` element specifies the name of the extended method. And, the `CodeReference` element specifies the static method. You can also add the `PSCodeMethod` element to the members of the `PSMemberSets` element.

XML

```
<Type>
  <Name>System.Xml.XmlNode</Name>
  <Members>
    <CodeMethod>
      <Name>ToString</Name>
      <CodeReference>
        <TypeName>Microsoft.PowerShell.ToStringCodeMethods</TypeName>
        <MethodName>XmlNode</MethodName>
      </CodeReference>
    </CodeMethod>
  </Members>
</Type>
```

Script methods

A script method defines a method whose value is the output of a script. In the following example, the `Convert.ToDateTime` method is added to the `System.Management.ManagementObject` type. The `PSScriptMethod` element defines the extended method as a script method. The `Name` element specifies the name of the extended method. And, the `Script` element specifies the script that generates the method value. You can also add the `PSScriptMethod` element to the members of the `PSMemberSets` element.

XML

```
<Type>
  <Name>System.Management.ManagementObject</Name>
  <Members>
    <ScriptMethod>
      <Name>Convert.ToDateTime</Name>
      <Script>

[System.Management.ManagementDateTimeConverter]::ToDateTIme($args[0])
      </Script>
    </ScriptMethod>
  </Members>
</Type>
```

See also

[Writing a Windows PowerShell Cmdlet](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Defining Default Member Sets for Objects

Article • 09/17/2021

The PSStandardMembers member set is used by Windows PowerShell to define the default property sets for an object. The default property sets can be used by commands such as the formatting cmdlets to display only those properties that are defined by the property set. The default property sets include DefaultDisplayProperty, DefaultDisplayPropertySet, and DefaultKeyPropertySet. Windows PowerShell ignores all other member sets and any other property sets added to the PSStandardMembers member set.

Member Set for System.Diagnostics.Process

In the following example, the PSStandardMembers member set defines the DefaultDisplayPropertySet property set for [System.Diagnostics.Process](#) objects. This property set is used by the [Format-List](#) cmdlet.

XML

```
<Type>
  <Name>System.Diagnostics.Process</Name>
  <Members>
    <MemberSet>
      <Name>PSStandardMembers</Name>
      <Members>
        <PropertySet>
          <Name>DefaultDisplayPropertySet</Name>
          <ReferencedProperties>
            <Name>Id</Name>
            <Name>Handles</Name>
            <Name>CPU</Name>
            <Name>Name</Name>
          </ReferencedProperties>
        </PropertySet>
      </Members>
    </MemberSet>
```

The following output shows the default properties returned by the [Format-List](#) cmdlet. Only the `Id`, `Handles`, `CPU`, and `Name` properties are returned for each process object.

PowerShell

```
Get-Process | Format-List
```

Output

```
Id      : 2036
Handles : 27
CPU     :
Name    : AEADISRV
```

```
Id      : 272
Handles : 38
CPU     :
Name    : agrsmsvc
...
```

See Also

[Writing a Windows PowerShell Cmdlet](#)

Custom Formatting Files

Article • 09/17/2021

The display format for the objects returned by cmdlets, functions, and scripts are defined using formatting files (`format.ps1xml` files). Several of these files are provided by Windows PowerShell to define the default display format for those objects returned by Windows PowerShell cmdlets. However, you can also create your own custom formatting files to overwrite the default display formats or to define the display of objects returned by your own commands.

Windows PowerShell uses the data in these formatting files to determine what is displayed and how the data is formatted. The displayed data can include the properties of an object or the value of a script block. Script blocks are used if you want to display some value that is not available directly from the properties of an object. For example, you may want to add the value of two properties of an object and display the sum as a separate piece of data. When you write your own formatting file, you will need to define *views* for the objects that you want to display. You can define a single view for each object, you can define a single view for multiple objects, or you can define multiple views for the same object. There is no limit to the number of views that you can define.

Important

Formatting files do not determine the elements of an object that are returned to the pipeline. When an object is returned to the pipeline, all members of that object are available.

Format Views

Formatting views can display objects in a table format, a list format, a wide format, and a custom format. For the most part, each formatting definition is described by a set of XML tags that describe a view. Each view contains the name of the view, the objects that use the view, and the elements of the view, such as the column and row information for a table view.

The following views are available.

Table view Lists the properties of an object or a script block value in one or more columns. Each column represents a property of the object or a script block value. You can define a table view that displays all the properties of an object, a subset of the properties of an object, or a combination of properties and script block values. Each row

of the table represents a returned object. For more information about this view, see [Table View](#).

List view Lists the properties of an object or a script block value in a single column. Each row of the list displays an optional label or the property name followed by the value of the property or script block. For more information about this view, see [List View](#).

Wide view Lists a single property of an object or a script block value in one or more columns. There is no label or header for this view. For more information about this view, see [Wide View](#).

Custom view Displays a customizable view of object properties or script block values that does not adhere to the rigid structure of table views, list views, or wide views. You can define a standalone custom view, or you can define a custom view that is used by another view, such as a table view or list view. For more information about this view, see [Custom View](#).

View XML Elements

The following example shows the XML tags used to define a table view that contains two columns. The [ViewDefinitions](#) element is the container element for all the views defined in the formatting file. The [View](#) element defines the specific table, list, wide, or custom view. Within each view, the [Name](#) element specifies the name of the view, the [ViewSelectedBy](#) element defines the objects that use the view, and the different control elements (such as the [TableControl](#) element) define the format of the view.

XML

```
ViewDefinitions
  <View>
    <Name>Name of View</Name>
    <ViewSelectedBy>
      <TypeName>Object to display using this view</TypeName>
      <TypeName>Object to display using this view</TypeName>
    </ViewSelectedBy>
    <TableControl>
      <TableHeaders>
        < TableColumnHeader>
          <Width></Width>
        </ TableColumnHeader>
        < TableColumnHeader>
          <Width></Width>
        </ TableColumnHeader>
      </TableHeaders>
      < TableRowEntries>
        < TableRowEntry>
          < TableColumnItems>
```

```
< TableColumnItem>
    <PropertyName>Header for column 1</PropertyName>
</TableColumnItem>
< TableColumnItem>
    <PropertyName>Header for column 2</PropertyName>
</TableColumnItem>
</TableColumnItems>
</TableRowEntry>
</TableRowEntries>
</TableControl>
</View>
</ViewDefinitions>
```

See Also

[Table View](#)

[List View](#)

[Wide View](#)

[Custom View](#)

[Writing a Windows PowerShell Cmdlet](#)

Requesting Confirmation

Article • 09/17/2021

This section discusses confirmation messages that can be displayed before a cmdlet, function, or provider performs an action.

In This Section

[Requesting Confirmation Process for Commands](#) Discusses the process that cmdlets, functions, and providers must follow to request a confirmation before they make a change to the system.

[Users Requesting Confirmation](#) Discusses how users can make a cmdlet, function, or provider request confirmation when the `System.Management.Automation.Cmdlet.ShouldProcess` method is called.

[Confirmation Messages](#) Provides samples of the different confirmation messages that can be displayed.

See Also

[Writing a Windows PowerShell Cmdlet](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Requesting Confirmation from Cmdlets

Article • 09/17/2021

Cmdlets should request confirmation when they are about to make a change to the system that is outside of the Windows PowerShell environment. For example, if a cmdlet is about to add a user account or stop a process, the cmdlet should require confirmation from the user before it proceeds. In contrast, if a cmdlet is about to change a Windows PowerShell variable, the cmdlet does not need to require confirmation.

In order to make a confirmation request, the cmdlet must indicate that it supports confirmation requests, and it must call the [System.Management.Automation.Cmdlet.ShouldProcess](#) and [System.Management.Automation.Cmdlet.ShouldContinue](#) (optional) methods to display a confirmation request message.

Supporting Confirmation Requests

To support confirmation requests, the cmdlet must set the `SupportsShouldProcess` parameter of the `Cmdlet` attribute to `true`. This enables the `Confirm` and `WhatIf` cmdlet parameters that are provided by Windows PowerShell. The `Confirm` parameter allows the user to control whether the confirmation request is displayed. The `WhatIf` parameter allows the user to determine whether the cmdlet should display a message or perform its action. Do not manually add the `Confirm` and `WhatIf` parameters to a cmdlet.

The following example shows a `Cmdlet` attribute declaration that supports confirmation requests.

C#

```
[Cmdlet(VerbsDiagnostic.Test, "RequestConfirmationTemplate1",
    SupportsShouldProcess = true)]
```

Calling the Confirmation request methods

In the cmdlet code, call the [System.Management.Automation.Cmdlet.ShouldProcess](#) method before the operation that changes the system is performed. Design the cmdlet so that if the call returns a value of `false`, the operation is not performed, and the cmdlet processes the next operation.

Calling the ShouldContinue Method

Most cmdlets request confirmation using only the [System.Management.Automation.Cmdlet.ShouldProcess](#) method. However, some cases might require additional confirmation. For these cases, supplement the [System.Management.Automation.Cmdlet.ShouldProcess](#) call with a call to the [System.Management.Automation.Cmdlet.ShouldContinue](#) method. This allows the cmdlet or provider to more finely control the scope of the **Yes to all** response to the confirmation prompt.

If a cmdlet calls the [System.Management.Automation.Cmdlet.ShouldContinue](#) method, the cmdlet must also provide a [Force](#) switch parameter. If the user specifies [Force](#) when the user invokes the cmdlet, the cmdlet should still call [System.Management.Automation.Cmdlet.ShouldProcess](#), but it should bypass the call to [System.Management.Automation.Cmdlet.ShouldContinue](#).

[System.Management.Automation.Cmdlet.ShouldContinue](#) will throw an exception when it is called from a non-interactive environment where the user cannot be prompted. Adding a [Force](#) parameter ensures that the command can still be performed when it is invoked in a non-interactive environment.

The following example shows how to call [System.Management.Automation.Cmdlet.ShouldProcess](#) and [System.Management.Automation.Cmdlet.ShouldContinue](#).

```
C#  
  
if (ShouldProcess (...))  
{  
    if (Force || ShouldContinue(...))  
    {  
        // Add code that performs the operation.  
    }  
}
```

The behavior of a [System.Management.Automation.Cmdlet.ShouldProcess](#) call can vary depending on the environment in which the cmdlet is invoked. Using the previous guidelines will help ensure that the cmdlet behaves consistently with other cmdlets, regardless of the host environment.

For an example of calling the [System.Management.Automation.Cmdlet.ShouldProcess](#) method, see [How to Request Confirmations](#).

Specify the Impact Level

When you create the cmdlet, specify the impact level (the severity) of the change. To do this, set the value of the `ConfirmImpact` parameter of the `Cmdlet` attribute to High, Medium, or Low. You can specify a value for `ConfirmImpact` only when you also specify the `SupportsShouldProcess` parameter for the cmdlet.

For most cmdlets, you do not have to explicitly specify `ConfirmImpact`. Instead, use the default setting of the parameter, which is Medium. If you set `ConfirmImpact` to High, the operation will be confirmed by default. Reserve this setting for highly disruptive actions, such as reformatting a hard-disk volume.

Calling Non-Confirmation Methods

If the cmdlet or provider must send a message but not request confirmation, it can call the following three methods. Avoid using the `System.Management.Automation.Cmdlet.WriteObject` method to send messages of these types because `System.Management.Automation.Cmdlet.WriteObject` output is intermingled with the normal output of your cmdlet or provider, which makes script writing difficult.

- To caution the user and continue with the operation, the cmdlet or provider can call the `System.Management.Automation.Cmdlet.WriteWarning` method.
- To provide additional information that the user can retrieve using the `Verbose` parameter, the cmdlet or provider can call the `System.Management.Automation.Cmdlet.WriteVerbose` method.
- To provide debugging-level detail for other developers or for product support, the cmdlet or provider can call the `System.Management.Automation.Cmdlet.WriteDebug` method. The user can retrieve this information using the `Debug` parameter.

Cmdlets and providers first call the following methods to request confirmation before they attempt to perform an operation that changes a system outside of Windows PowerShell:

- `System.Management.Automation.Cmdlet.ShouldProcess`
- `System.Management.Automation.Provider.CmdletProvider.ShouldProcess`

They do so by calling the `System.Management.Automation.Cmdlet.ShouldProcess` method, which prompts the user to confirm the operation based on how the user invoked the command.

See Also

Writing a Windows PowerShell Cmdlet

Users Requesting Confirmation

Article • 09/24/2024

When you specify a value of `true` for the `SupportsShouldProcess` parameter of the Cmdlet attribute declaration, the **Confirm** parameter is added to the parameters of the cmdlet.

In the default environment, users can specify the **Confirm** parameter so that confirmation is requested when the `ShouldProcess()` method is called. This forces confirmation regardless of the impact level setting.

If **Confirm** parameter is not used, the `ShouldProcess()` call requests confirmation if the `ConfirmImpact` setting is equal to or greater than the `$ConfirmPreference` preference variable. The default setting of `$ConfirmPreference` is **High**. Therefore, in the default environment, only cmdlets and providers that specify a high-impact action request confirmation.

If **Confirm** is explicitly set to false (`-Confirm:$false`), the cmdlet runs without prompting for confirmation and the `$ConfirmPreference` shell variable is ignored.

Remarks

- For cmdlets and providers that specify `SupportsShouldProcess`, but not `ConfirmImpact`, those actions are handled as `Medium` impact actions, and they will not prompt by default. Their impact level is less than the default setting of the `$ConfirmPreference` preference variable.
- If the user specifies the `Verbose` parameter, they will be notified of the operation even if they are not prompted for confirmation.

See Also

- [Writing a Windows PowerShell Cmdlet](#)
- [System.Management.Automation.Cmdlet.ShouldProcess](#)

Confirmation Messages

Article • 03/24/2025

Here are different confirmation messages that can be displayed depending on the variants of the [System.Management.Automation.Cmdlet.ShouldProcess](#) and [System.Management.Automation.Cmdlet.ShouldContinue](#) methods that are called.

ⓘ Important

For sample code that shows how to request confirmations, see [How to Request Confirmations](#).

Specifying the Resource

You can specify the resource that is about to be changed by calling the [System.Management.Automation.Cmdlet.ShouldProcess](#) method. In this case, you supply the resource by using the `target` parameter of the method, and the operation is added by Windows PowerShell. In the following message, the text "MyResource" is the resource acted on and the operation is the name of the command that makes the call.

Output

```
Confirm
Are you sure you want to perform this action?
Performing operation "Test-RequestConfirmationTemplate1" on Target
"MyResource".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):
```

If the user selects **Yes** or **Yes to All** to the confirmation request (as shown in the following example), a call to the [System.Management.Automation.Cmdlet.ShouldContinue](#) method is made, which causes a second confirmation message to be displayed.

Output

```
Confirm
Are you sure you want to perform this action?
Performing operation "Test-RequestConfirmationTemplate1" on Target
"MyResource".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"): y
```

```
Confirm  
Continue with this operation?  
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"):
```

Specifying the Operation and Resource

You can specify the resource that is about to be changed and the operation that the command is about to perform by calling the [System.Management.Automation.Cmdlet.ShouldProcess](#) method. In this case, you supply the resource by using the `target` parameter and the operation by using the `target` parameter. In the following message, the text "MyResource" is the resource acted on and "MyAction" is the operation to be performed.

Output

```
Confirm  
Are you sure you want to perform this action?  
Performing operation "MyAction" on Target "MyResource".  
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"):
```

If the user selects **Yes** or **Yes to All** to the previous message, a call to the [System.Management.Automation.Cmdlet.ShouldContinue](#) method is made, which causes a second confirmation message to be displayed.

Output

```
Confirm  
Are you sure you want to perform this action?  
Performing operation "MyAction" on Target "MyResource".  
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"): y  
  
Confirm  
Continue with this operation?  
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"):
```

See Also

[Writing a Windows PowerShell Cmdlet](#)

Windows PowerShell Error Reporting

Article • 09/17/2021

The topics in this section discuss how cmdlets report errors.

In This Section

[Error Reporting Concepts](#) Describes the two mechanisms that cmdlets can use to report errors.

[Terminating Errors](#) Describes the method used to report terminating errors, where that method can be called from within the cmdlet, and exceptions that can be returned by the Windows PowerShell runtime when the method is called.

[Non-Terminating Errors](#) Describes the method used to report non-terminating errors and where that method can be called from within the cmdlet.

[Displaying Error Information by Category](#) Discusses the ways that users can display error.

[Windows PowerShell Error Records](#) Describes the components of an error record.

[Interpreting ErrorRecord Objects](#) Discusses how ErrorRecord objects are interpreted.

See Also

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Error Reporting Concepts

Article • 03/24/2025

Windows PowerShell provides two mechanisms for reporting errors: one mechanism for *terminating errors* and another mechanism for *non-terminating errors*. It is important for your cmdlet to report errors correctly so that the host application that is running your cmdlets can react in an appropriate manner.

Your cmdlet should call the

[System.Management.Automation.Cmdlet.ThrowTerminatingError*](#) method when an error occurs that does not or should not allow the cmdlet to continue to process its input objects. Your cmdlet should call the [System.Management.Automation.Cmdlet.WriteError](#) method to report non-terminating errors when the cmdlet can continue processing the input objects. Both methods provide an error record that the host application can use to investigate the cause of the error.

Use the following guidelines to determine whether an error is a terminating or non-terminating error.

- An error is a terminating error if it prevents your cmdlet from continuing to process the current object or from successfully processing any further input objects, regardless of their content.
- An error is a terminating error if you do not want your cmdlet to continue processing the current object or any further input objects, regardless of their content.
- An error is a terminating error if it occurs in a cmdlet that does not accept or return an object or if it occurs in a cmdlet that accepts or returns only one object.
- An error is a non-terminating error if you want your cmdlet to continue processing the current object and any further input objects.
- An error is a non-terminating error if it is related to a specific input object or subset of input objects.

See Also

[System.Management.Automation.Cmdlet.ThrowTerminatingError*](#)

[System.Management.Automation.Cmdlet.WriteError](#)

[Windows PowerShell Error Records](#)

Writing a Windows PowerShell Cmdlet

Terminating Errors

Article • 03/24/2025

This topic discusses the method used to report terminating errors. It also discusses how to call the method from within the cmdlet, and it discusses the exceptions that can be returned by the Windows PowerShell runtime when the method is called.

When a terminating error occurs, the cmdlet should report the error by calling the [System.Management.Automation.Cmdlet.ThrowTerminatingError*](#) method. This method allows the cmdlet to send an error record that describes the condition that caused the terminating error. For more information about error records, see [Windows PowerShell Error Records](#).

When the [System.Management.Automation.Cmdlet.ThrowTerminatingError*](#) method is called, the Windows PowerShell runtime permanently stops the execution of the pipeline and throws a [System.Management.Automation.PipelineStoppedException](#) exception. Any subsequent attempts to call [System.Management.Automation.Cmdlet.WriteObject](#), [System.Management.Automation.Cmdlet.WriteError](#), or several other APIs causes those calls to throw a [System.Management.Automation.PipelineStoppedException](#) exception.

The [System.Management.Automation.PipelineStoppedException](#) exception can also occur if another cmdlet in the pipeline reports a terminating error, if the user has asked to stop the pipeline, or if the pipeline has been halted before completion for any reason. The cmdlet does not need to catch the [System.Management.Automation.PipelineStoppedException](#) exception unless it must clean up open resources or its internal state.

Cmdlets can write any number of output objects or non-terminating errors before reporting a terminating error. However, the terminating error permanently stops the pipeline, and no further output, terminating errors, or non-terminating errors can be reported.

Cmdlets can call [System.Management.Automation.Cmdlet.ThrowTerminatingError*](#) only from the thread that called the [System.Management.Automation.Cmdlet.BeginProcessing](#), [System.Management.Automation.Cmdlet.ProcessRecord](#), or [System.Management.Automation.Cmdlet.EndProcessing](#) input processing method. Do not attempt to call [System.Management.Automation.Cmdlet.ThrowTerminatingError*](#) or [System.Management.Automation.Cmdlet.WriteError](#) from another thread. Instead, errors must be communicated back to the main thread.

It is possible for a cmdlet to throw an exception in its implementation of the [System.Management.Automation.Cmdlet.BeginProcessing](#), [System.Management.Automation.Cmdlet.ProcessRecord](#), or [System.Management.Automation.Cmdlet.EndProcessing](#) method. Any exception thrown from these methods (except for a few severe error conditions that stop the Windows PowerShell host) is interpreted as a terminating error which stops the pipeline, but not Windows PowerShell as a whole. (This applies only to the main cmdlet thread. Uncaught exceptions in threads spawned by the cmdlet, in general, halt the Windows PowerShell host.) We recommend that you use

[System.Management.Automation.Cmdlet.ThrowTerminatingError*](#) rather than throwing an exception because the error record provides additional information about the error condition, which is useful to the end-user. Cmdlets should honor the managed code guideline against catching and handling all exceptions (`catch (Exception e)`). Convert only exceptions of known and expected types into error records.

See Also

[System.Management.Automation.Cmdlet.BeginProcessing](#)

[System.Management.Automation.Cmdlet.EndProcessing](#)

[System.Management.Automation.Cmdlet.ProcessRecord](#)

[System.Management.Automation.PipelineStoppedException](#)

[System.Management.Automation.Cmdlet.ThrowTerminatingError*](#)

[System.Management.Automation.Cmdlet.WriteError](#)

[Windows PowerShell Error Records](#)

[Writing a Windows PowerShell Cmdlet](#)

Non-Terminating Errors

Article • 09/17/2021

This topic discusses the method used to report non-terminating errors. It also discusses how to call the method from within the cmdlet.

When a non-terminating error occurs, the cmdlet should report this error by calling the [System.Management.Automation.Cmdlet.WriteError](#) method. When the cmdlet reports a non-terminating error, the cmdlet can continue to operate on this input object and on further incoming pipeline objects. If the cmdlet calls the [System.Management.Automation.Cmdlet.WriteError](#) method, the cmdlet can write an error record that describes the condition that caused the non-terminating error. For more information about error records, see [Windows PowerShell Error Records](#).

Cmdlets can call [System.Management.Automation.Cmdlet.WriteError](#) as necessary from within their input processing methods. However, cmdlets can call [System.Management.Automation.Cmdlet.WriteError](#) only from the thread that called the [System.Management.Automation.Cmdlet.BeginProcessing](#), [System.Management.Automation.Cmdlet.ProcessRecord](#), or [System.Management.Automation.Cmdlet.EndProcessing](#) input processing method. Do not call [System.Management.Automation.Cmdlet.WriteError](#) from another thread. Instead, communicate errors back to the main thread.

See Also

[System.Management.Automation.Cmdlet.WriteError](#)

[System.Management.Automation.Cmdlet.BeginProcessing](#)

[System.Management.Automation.Cmdlet.ProcessRecord](#)

[System.Management.Automation.Cmdlet.EndProcessing](#)

[Windows PowerShell Error Records](#)

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on
GitHub



PowerShell feedback

PowerShell is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Displaying Error Information

Article • 09/17/2021

This topic discusses the ways in which users can display error information.

When your cmdlet encounters an error, the presentation of the error information will, by default, resemble the following error output.

PowerShell

```
$ Stop-Service lanmanworkstation  
You do not have sufficient permissions to stop the service Workstation.
```

However, users can view errors by category by setting the `$ErrorView` variable to `"CategoryView"`. Category view displays specific information from the error record rather than a free-text description of the error. This view can be useful if you have a long list of errors to scan. In category view, the previous error message is displayed as follows.

PowerShell

```
$ $ErrorView = "CategoryView"  
$ Stop-Service lanmanworkstation  
CloseError: (System.ServiceProcess.ServiceController:ServiceController)  
[Stop-Service], ServiceCommandException
```

For more information about error categories, see [Windows PowerShell Error Records](#).

See Also

[Windows PowerShell Error Records](#)

[Writing a Windows PowerShell Cmdlet](#)

Windows PowerShell Error Records

Article • 09/17/2021

Cmdlets must pass an [System.Management.Automation.ErrorRecord](#) object that identifies the error condition for terminating and non-terminating errors.

The [System.Management.Automation.ErrorRecord](#) object contains the following information:

- The exception that describes the error. Often, this is an exception that the cmdlet caught and converted into an error record. Every error record must contain an exception.

If the cmdlet did not catch an exception, it must create a new exception and choose the exception class that best describes the error condition. However, you do not need to throw the exception because it can be accessed through the [System.Management.Automation.ErrorRecord.Exception](#) property of the [System.Management.Automation.ErrorRecord](#) object.

- An error identifier that provides a targeted designator that can be used for diagnostic purposes and by Windows PowerShell scripts to handle specific error conditions with specific error handlers. Every error record must contain an error identifier (see Error Identifier).
- An error category that provides a general designator that can be used for diagnostic purposes. Every error record must specify an error category (see Error Category).
- An optional replacement error message and a recommended action (see Replacement Error Message).
- Optional invocation information about the cmdlet that threw the error. This information is specified by Windows PowerShell (see Invocation Message).
- The target object that was being processed when the error occurred. This might be the input object, or it might be another object that your cmdlet was processing. For example, for the command `Remove-Item -Recurse C:\somedirectory`, the error might be an instance of a FileInfo object for "C:\somedirectory\lockedfile". The target object information is optional.

Error Identifier

When you create an error record, specify an identifier that designates the error condition within your cmdlet. Windows PowerShell combines the targeted identifier with the name of your cmdlet to create a fully qualified error identifier. The fully qualified error identifier can be accessed through the

[System.Management.Automation.ErrorRecord.FullyQualifiedErrorId](#) property of the [System.Management.Automation.ErrorRecord](#) object. The error identifier is not available by itself. It is available only as part of the fully qualified error identifier.

Use the following guidelines to generate error identifiers when you create error records:

- Make error identifiers specific to an error condition. Target the error identifiers for diagnostic purposes and for scripts that handle specific error conditions with specific error handlers. A user should be able to use the error identifier to identify the error and its source. Error identifiers also enable reporting for specific error conditions from existing exceptions so that new exception subclasses are not required.
- In general, assign different error identifiers to different code paths. The end-user benefits from specific identifiers. Often, each code path that calls [System.Management.Automation.Cmdlet.WriteError](#) or [System.Management.Automation.Cmdlet.ThrowTerminatingError*](#) has its own identifier. As a rule, define a new identifier when you define a new template string for the error message, and vice-versa. Do not use the error message as an identifier.
- When you publish code using a particular error identifier, you establish the semantics of errors with that identifier for your complete product support lifecycle. Do not reuse it in a context that is semantically different from the original context. If the semantics of this error change, create and then use a new identifier.
- You should generally use a particular error identifier only for exceptions of a particular CLR type. If the type of the exception or the type of the target object changes, create and then use a new identifier.
- Choose text for your error identifier that concisely corresponds to the error that you are reporting. Use standard .NET Framework naming and capitalization conventions. Do not use white space or punctuation. Do not localize error identifiers.
- Do not dynamically generate error identifiers in a non-reproducible way. For example, do not incorporate error information such as a process ID. Error identifiers are useful only if they correspond to the error identifiers seen by other users who are experiencing the same error condition.

Error Category

When you create an error record, specify the category of the error using one of the constants defined by the [System.Management.Automation.ErrorCategory](#) enumeration. Windows PowerShell uses the error category to display error information when users set the `$ErrorView` variable to `"CategoryView"`.

Avoid using the [System.Management.Automation.ErrorCategory](#) **NotSpecified** constant. If you have any information about the error or about the operation that caused the error, choose the category that best describes the error or the operation, even if the category is not a perfect match.

The information displayed by Windows PowerShell is referred to as the category-view string and is built from the properties of the [System.Management.Automation.ErrorCategoryInfo](#) class. (This class is accessed through the error [System.Management.Automation.ErrorRecord.CategoryInfo](#) property.)

```
{Category}: ({TargetName}:{TargetType}):[{Activity}], {Reason}
```

The following list describes the information displayed:

- Category: A Windows PowerShell-defined [System.Management.Automation.ErrorCategory](#) constant.
- TargetName: By default, the name of the object the cmdlet was processing when the error occurred. Or, another cmdlet-defined string.
- TargetType: By default, the type of the target object. Or, another cmdlet-defined string.
- Activity: By default, the name of the cmdlet that created the error record. Or, some other cmdlet-defined string.
- Reason: By default, the exception type. Or, another cmdlet-defined string.

Replacement Error Message

When you develop an error record for a cmdlet, the default error message for the error comes from the default message text in the [System.Exception.Message](#) property. This is a read-only property whose message text is intended only for debugging purposes (according to the .NET Framework guidelines). We recommend that you create an error

message that replaces or augments the default message text. Make the message more user-friendly and more specific to the cmdlet.

The replacement message is provided by an [System.Management.Automation.ErrorDetails](#) object. Use one of the following constructors of this object because they provide additional localization information that can be used by Windows PowerShell.

- [ErrorDetails\(Cmdlet, String, String, Object\[\]\)](#): Use this constructor if your template string is a resource string in the same assembly in which the cmdlet is implemented or if you want to load the template string through an override of the [System.Management.Automation.Cmdlet.GetResourceString](#) method.
- [ErrorDetails\(Assembly, String, String, Object\[\]\)](#): Use this constructor if the template string is in another assembly and you do not load it through an override of [System.Management.Automation.Cmdlet.GetResourceString](#).

The replacement message should conform to the .NET Framework design guidelines for writing exception messages with a small difference. The guidelines state that exception messages should be written for developers. These replacement messages should be written for the cmdlet user.

The replacement error message must be added before the [System.Management.Automation.Cmdlet.WriteError](#) or [System.Management.Automation.Cmdlet.ThrowTerminatingError*](#) methods are called. To add a replacement message, set the [System.Management.Automation.ErrorRecord.ErrorDetails](#) property of the error record. When this property is set, Windows PowerShell displays the [System.Management.Automation.ErrorDetails.Message*](#) property instead of the default message text.

Recommended Action Information

The [System.Management.Automation.ErrorDetails](#) object can also provide information about what actions are recommended when the error occurs.

Invocation information

When a cmdlet uses [System.Management.Automation.Cmdlet.WriteError](#) or [System.Management.Automation.Cmdlet.ThrowTerminatingError*](#) to report an error record, Windows PowerShell automatically adds information that describes the command that was invoked when the error occurred. This information is provided by a

[System.Management.Automation.InvocationInfo](#) object that contains the name of the cmdlet that was invoked by the command, the command itself, and information about the pipeline or script. This property is read-only.

See Also

[System.Management.Automation.Cmdlet.WriteError](#)

[System.Management.Automation.Cmdlet.ThrowTerminatingError*](#)

[System.Management.Automation.ErrorCategory](#)

[System.Management.Automation.ErrorCategoryInfo](#)

[System.Management.Automation.ErrorRecord](#)

[System.Management.Automation.ErrorDetails](#)

[System.Management.Automation.InvocationInfo](#)

[Windows PowerShell Error Reporting](#)

[Writing a Windows PowerShell Cmdlet](#)

Interpreting ErrorRecord Objects

Article • 09/15/2023

In most cases, an [System.Management.Automation.ErrorRecord](#) object represents a non-terminating error generated by a command or script. Terminating errors can also specify the additional information in an ErrorRecord via the [System.Management.Automation.IContainsErrorRecord](#) interface.

If you want to write an error handler in your script or a host to handle specific errors that occur during command or script execution, you must interpret the [System.Management.Automation.ErrorRecord](#) object to determine whether it represents the class of error that you want to handle.

When a cmdlet encounters a terminating or non-terminating error, it should create an error record that describes the error condition. The host application must investigate these error records and perform whatever action will mitigate the error. The host application must also investigate error records for non-terminating errors that failed to process a record but were able to continue, and it must investigate error records for terminating errors that caused the pipeline to stop.

ⓘ Note

For terminating errors, the cmdlet calls the [System.Management.Automation.Cmdlet.ThrowTerminatingError](#) method. For non-terminating errors, the cmdlet calls the [System.Management.Automation.Cmdlet.WriteError](#) method.

Error Record Design

Error records are designed to provide additional error information that is not available in exceptions while ensuring that the combined information in each error record is unique. This uniqueness allows the host application to inspect the different parts of the error record so that it can identify the error condition and the action the host must take.

Interpreting Error Records

You can review several parts of the error record to identify the error. These parts include the following:

- The error category
- The error exception
- The fully qualified error identifier (FQID)
- Other information

The Error Category

The error category of the error record is one of the predefined constants provided by the [System.Management.Automation.ErrorCategory](#) enumeration. This information is available through the [System.Management.Automation.ErrorRecord.CategoryInfo](#) property of the [System.Management.Automation.ErrorRecord](#) object.

The cmdlet can specify the CloseError, OpenError, InvalidType, ReadError, and WriteError categories, and other error categories. The host application can use the error category to capture groups of errors.

The Exception

The exception included in the error record is provided by the cmdlet and can be accessed through the [System.Management.Automation.ErrorRecord.Exception](#) property of the [System.Management.Automation.ErrorRecord](#) object.

Host applications can use the `is` keyword to identify that the exception is of a specific class or of a derived class. It is better to branch on the exception type, as shown in the following example.

PowerShell

```
`if (MyNonTerminatingError.Exception is AccessDeniedException)`  
{  
    ...  
}
```

This way, you catch the derived classes. However, there are problems if the exception is serialized.

The FQID

The FQID is the most specific information you can use to identify the error. It is a string that includes a cmdlet-defined identifier, the name of the cmdlet class, and the source

that reported the error. In general, an error record is analogous to an event record of a Windows Event log. The FQID is analogous to the following tuple, which identifies the class of the event record: (*log name*, *source*, *event ID*).

The FQID is designed to be inspected as a single string. However, cases exist in which the error identifier is designed to be parsed by the host application. The following example is a well-formed fully qualified error identifier.

```
CommandNotFoundException,Microsoft.PowerShell.Commands.GetCommandCommand.
```

In the previous example, the first token is the error identifier, which is followed by the name of the cmdlet class. The error identifier can be a single token, or it can be a dot-separated identifier that allows for branching on inspection of the identifier. Do not use white space or punctuation in the error identifier. It is especially important not to use a comma; a comma is used by Windows PowerShell to separate the identifier and the cmdlet class name.

Other Information

The [System.Management.Automation.ErrorRecord](#) object might also provide information that describes the environment in which the error occurred. This information includes items such as error details, invocation information, and the target object that was being processed when the error occurred. Although this information might be useful to the host application, it is not typically used to identify the error. This information is available through the following properties:

- [System.Management.Automation.ErrorRecord.ErrorDetails](#)
- [System.Management.Automation.ErrorRecord.InvocationInfo](#)
- [System.Management.Automation.ErrorRecord.TargetObject](#)

See Also

- [System.Management.Automation.ErrorRecord](#)
- [System.Management.Automation.ErrorCategory](#)
- [System.Management.Automation.ErrorCategoryinfo](#)
- [System.Management.Automation.Cmdlet.WriteError](#)
- [System.Management.Automation.Cmdlet.ThrowTerminatingError*](#)

- Adding Non-Terminating Error Reporting to Your Cmdlet
- Windows PowerShell Error Reporting
- Writing a Windows PowerShell Cmdlet

Background Jobs

Article • 09/17/2021

Cmdlets can perform their action internally or as a Windows PowerShell *background job*. When a cmdlet runs as a background job, the work is done asynchronously in its own thread separate from the pipeline thread that the cmdlet is using. From the user perspective, when a cmdlet runs as a background job, the command prompt returns immediately even if the job takes an extended amount of time to complete, and the user can continue without interruption while the job runs.

Background Jobs, Child Jobs, and the Job Repository

The job object that is returned by the cmdlets that support background jobs defines the job. (The [Start-Job](#) cmdlet also returns a job object.) The name of the job, an identifier that is used to specify the job, the state information, and the child jobs are included in this definition. The job does not perform any of the work. Each background job has at least one child job because the child job performs the actual work. When you run a cmdlet so that the work is performed as a background job, the cmdlet must add the job and the child jobs to a common repository, referred to as the *job repository*.

For more information about how background jobs are handled at the command line, see the following:

- [about_Jobs](#)
- [about_Job_Details](#)
- [about_Remote_Jobs](#)

Writing a Cmdlet That Runs as a Background Job

To write a cmdlet that can be run as a background job, you must complete the following tasks:

- Define an `asJob` switch parameter so that the user can decide whether to run the cmdlet as a background job.

- Create an object that derives from the [System.Management.Automation.Job](#) class. This object can be a custom job object or a job object provided by Windows PowerShell, such as a [System.Management.Automation.PSEventJob](#) object.
- In a record processing method, add an `if` statement that detects whether the cmdlet should run as a background job.
- For custom job objects, implement the job class.
- Return the appropriate objects, depending on whether the cmdlet is run as a background job.

For a code example, see [How to Support Jobs](#).

Background Job-Related APIs

The following APIs are provided by Windows PowerShell to manage background jobs.

[System.Management.Automation.Job](#) Derives custom job objects. This is an abstract class.

[System.Management.Automation.JobRepository](#) Manages and provides information about the current active background jobs.

[System.Management.Automation.JobState](#) Defines the state of the background job. States include Started, Running, and Stopped.

[System.Management.Automation.JobStateInfo](#) Provides information about the state of a background job and, if the last state change was caused by an error, the reason the job entered its current state.

[System.Management.Automation.JobStateEventArgs](#) Provides the arguments for an event that is raised when a background job changes state.

Windows PowerShell Job Cmdlets

The following cmdlets are provided by Windows PowerShell to manage background jobs.

[Get-Job](#)

Gets Windows PowerShell background jobs that are running in the current session.

[Receive-Job](#)

Gets the results of the Windows PowerShell background jobs in the current session.

[Remove-Job](#)

Deletes a Windows PowerShell background job.

[Start-Job](#)

Starts a Windows PowerShell background job.

[Stop-Job](#)

Stops a Windows PowerShell background job.

[Wait-Job](#)

Suppresses the command prompt until one or all of the Windows PowerShell background jobs running in the session are complete.

See Also

[Writing a Windows PowerShell Cmdlet](#)

Invoking Cmdlets and Scripts Within a Cmdlet

Article • 12/18/2023

A cmdlet can invoke other cmdlets and scripts from within the input processing method of the cmdlet. This allows you to add the functionality of existing cmdlets and scripts to your cmdlet without having to rewrite the code.

The Invoke Method

All cmdlets can invoke an existing cmdlet by calling the [System.Management.Automation.Cmdlet.Invoke](#) method from within an input processing method, such as [System.Management.Automation.Cmdlet.BeginProcessing](#), that is overridden by the cmdlet. However, you can invoke only those cmdlets that derive directly from the [System.Management.Automation.Cmdlet](#) class. You cannot invoke a cmdlet that derives from the [System.Management.Automation.PSCmdlet](#) class.

The [System.Management.Automation.Cmdlet.Invoke*](#) method has the following variants.

[System.Management.Automation.Cmdlet.Invoke](#) This variant invokes the cmdlet object and returns a collection of "T" type objects.

[System.Management.Automation.Cmdlet.Invoke](#) This variant invokes the cmdlet object and returns a strongly typed enumerator. This variant allows the user to use the objects in the collection to perform custom operations.

Examples

[] [Expand table](#)

Example	Description
Invoking Cmdlets Within a Cmdlet	This example shows how to invoke a cmdlet from within another cmdlet.
Invoking Scripts Within a Cmdlet	This example shows how to invoke a script that is supplied to the cmdlet from within another cmdlet.

See Also

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Cmdlet Sets

Article • 03/24/2025

When you design your cmdlets, you might encounter cases in which you need to perform several actions on the same piece of data. For example, you might need to get and set data or start and stop a process. Although you will need to create separate cmdlets to perform each action, your cmdlet design should include a base class from which the classes for the individual cmdlets are derived.

Keep the following things in mind when implementing a base class.

- Declare any common parameters used by all the derived cmdlets in the base class.
- Add cmdlet-specific parameters to the appropriate cmdlet class.
- Override the appropriate input processing method in the base class.
- Declare the [System.Management.Automation.CmdletAttribute](#) attribute on all cmdlet classes, but do not declare it on the base class.
- Implement a [System.Management.Automation.PSSnapIn](#) or [System.Management.Automation.CustomPSSnapIn](#) class whose name and description reflects the set of cmdlets.

Example

The following example shows the implementation of a base class that is used by Get-Proc and Stop-Proc cmdlet that derive from the same base class.

C#

```
using System;
using System.Diagnostics;
using System.Management.Automation;           //Windows PowerShell
namespace.

namespace Microsoft.Samples.PowerShell.Commands
{

    #region ProcessCommands

    /// <summary>
    /// This class implements a Stop-Proc cmdlet. The parameters
    /// for this cmdlet are defined by the BaseProcCommand class.
    /// </summary>
    [Cmdlet(VerbsLifecycle.Stop, "Proc", SupportsShouldProcess = true)]
```

```

public class StopProcCommand : BaseProcCommand
{
    public override void ProcessObject(Process process)
    {
        if (ShouldProcess(process.ProcessName, "Stop-Proc"))
        {
            process.Kill();
        }
    }
}

/// <summary>
/// This class implements a Get-Proc cmdlet. The parameters
/// for this cmdlet are defined by the BaseProcCommand class.
/// </summary>

[Cmdlet(VerbsCommon.Get, "Proc")]
public class GetProcCommand : BaseProcCommand
{
    public override void ProcessObject(Process process)
    {
        WriteObject(process);
    }
}

/// <summary>
/// This class is the base class that defines the common
/// functionality used by the Get-Proc and Stop-Proc
/// cmdlets.
/// </summary>
public class BaseProcCommand : Cmdlet
{
    #region Parameters

    // Defines the Name parameter that is used to
    // specify a process by its name.
    [Parameter()
        Position = 0,
        Mandatory = true,
        ValueFromPipeline = true,
        ValueFromPipelineByPropertyName = true
    ]
    public string[] Name
    {
        get { return processNames; }
        set { processNames = value; }
    }
    private string[] processNames;

    // Defines the Exclude parameter that is used to
    // specify which processes should be excluded when
    // the cmdlet performs its action.
    [Parameter()]
    public string[] Exclude
    {

```

```

        get { return excludeNames; }
        set { excludeNames = value; }
    }
    private string[] excludeNames = new string[0];
#endregion Parameters

    public virtual void ProcessObject(Process process)
    {
        throw new NotImplementedException("This method should be
overridden.");
    }

#region Cmdlet Overrides
// <summary>
// For each of the requested process names, retrieve and write
// the associated processes.
// </summary>
protected override void ProcessRecord()
{
    // Set up the wildcard characters used in resolving
    // the process names.
    WildcardOptions options = WildcardOptions.IgnoreCase |
                                WildcardOptions.Compiled;

    WildcardPattern[] include = new WildcardPattern[Name.Length];
    for (int i = 0; i < Name.Length; i++)
    {
        include[i] = new WildcardPattern(Name[i], options);
    }

    WildcardPattern[] exclude = new WildcardPattern[Exclude.Length];
    for (int i = 0; i < Exclude.Length; i++)
    {
        exclude[i] = new WildcardPattern(Exclude[i], options);
    }

    foreach (Process p in Process.GetProcesses())
    {
        foreach (WildcardPattern wIn in include)
        {
            if (wIn.IsMatch(p.ProcessName))
            {
                bool processThisOne = true;
                foreach (WildcardPattern wOut in exclude)
                {
                    if (wOut.IsMatch(p.ProcessName))
                    {
                        processThisOne = false;
                        break;
                    }
                }
                if (processThisOne)
                {
                    ProcessObject(p);
                }
            }
        }
    }
}

```

```
        break;
    }
}
}
#endregion Cmdlet Overrides
}
#endregion ProcessCommands
}
```

See Also

[Writing a Windows PowerShell Cmdlet](#)

Windows PowerShell Session State

Article • 09/17/2021

Session state refers to the current configuration of a Windows PowerShell session or module. A Windows PowerShell session is the operational environment that is used interactively by the command-line user or programmatically by a host application. The session state for a session is referred to as the global session state.

From a developer perspective, a Windows PowerShell session refers to the time between when a host application opens a Windows PowerShell runspace and when it closes the runspace. Looked at another way, the session is the lifetime of an instance of the Windows PowerShell engine that is invoked while the runspace exists.

Module Session State

Module session states are created whenever the module or one of its nested modules is imported into the session. When a module exports an element such as a cmdlet, function, or script, a reference to that element is added to the global session state of the session. However, when the element is run, it is executed within the session state of the module.

Session-State Data

Session state data can be public or private. Public data is available to calls from outside the session state while private data is available only to calls from within the session state. For example, a module can have a private function that can be called only by the module or only internally by a public element that has been exported. This is similar to the private and public members of a .NET Framework type.

Session-state data is stored by the current instance of the execution engine within the context of the current Windows PowerShell session. Session-state data consists of the following items:

- Path information
- Drive information
- Windows PowerShell provider information
- Information about the imported modules and references to the module elements (such as cmdlets, functions, and scripts) that are exported by the module. This

information and these references are for the global session state only.

- Session-state variable information

Accessing Session-State Data Within Cmdlets

Cmdlets can access session-state data either indirectly through the [System.Management.Automation.PSCmdlet.SessionState*](#) property of the cmdlet class or directly through the [System.Management.Automation.SessionState](#) class. The [System.Management.Automation.SessionState](#) class provides properties that can be used to investigate different types of session-state data.

See Also

[System.Management.Automation.PSCmdlet.SessionState](#)

[System.Management.Automation.SessionState](#)

[Windows PowerShell Cmdlets](#)

[Writing a Windows PowerShell Cmdlet](#)

[Windows PowerShell Shell SDK](#)

Examples of Cmdlet Code

Article • 09/17/2021

This section contains examples of cmdlet code that you can use to start writing your own cmdlets.

ⓘ Important

If you want step-by-step instructions for writing cmdlets, see [Tutorials for Writing Cmdlets](#).

In This Section

[How to Write a Simple Cmdlet](#) This example shows the basic structure of cmdlet code.

[How to Declare Cmdlet Parameters](#) This example shows how to declare the different types of parameters.

[How to Declare Parameter Sets](#) This example shows how to declare sets of parameters that can change the action a cmdlet performs.

[How to Validate Parameter Input](#) These examples show how to validate parameter input.

[How to Declare Dynamic Parameters](#) This example shows how to declare a parameter that is added at runtime.

[How to Invoke Scripts Within a Cmdlet](#) This example shows how to invoke a script that is supplied to a cmdlet.

[How To Override Input Processing Methods](#) These examples show the basic structure used to override the BeginProcessing, ProcessRecord, and EndProcessing methods.

[How to Support ShouldProcess Calls](#) This example shows how the `System.Management.Automation.Cmdlet.ShouldProcess` and `System.Management.Automation.Cmdlet.ShouldContinue` methods should be called from within a cmdlet.

[How to Support Transactions](#) This example shows how to indicate that the cmdlet supports transactions and how to implement the action that is taken when the cmdlet is used within a transaction.

[How to Support Jobs](#) This example shows how to support jobs when you write cmdlets.

[How to Invoke a Cmdlet From Within a Cmdlet](#) This example shows how to invoke a cmdlet from within another cmdlet, which allows you to add the functionality of the invoked cmdlet to the cmdlet you are developing.

See Also

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

How to write a cmdlet

Article • 09/17/2021

This article shows how to write a cmdlet. The `Send-Greeting` cmdlet takes a single user name as input and then writes a greeting to that user. Although the cmdlet does not do much work, this example demonstrates the major sections of a cmdlet.

Steps to write a cmdlet

1. To declare the class as a cmdlet, use the **Cmdlet** attribute. The **Cmdlet** attribute specifies the verb and the noun for the cmdlet name.

For more information about the **Cmdlet** attribute, see [CmdletAttribute Declaration](#).

2. Specify the name of the class.

3. Specify that the cmdlet derives from either of the following classes:

- [System.Management.Automation.Cmdlet](#)
- [System.Management.Automation.PSCmdlet](#)

4. To define the parameters for the cmdlet, use the **Parameter** attribute. In this case, only one required parameter is specified.

For more information about the **Parameter** attribute, see [ParameterAttribute Declaration](#).

5. Override the input processing method that processes the input. In this case, the [System.Management.Automation.Cmdlet.ProcessRecord](#) method is overridden.

6. To write the greeting, use the method

[System.Management.Automation.Cmdlet.WriteObject](#). The greeting is displayed in the following format:

Output

Hello <UserName>!

Example

C#

```
using System.Management.Automation; // Windows PowerShell assembly.

namespace SendGreeting
{
    // Declare the class as a cmdlet and specify the
    // appropriate verb and noun for the cmdlet name.
    [Cmdlet(VerbsCommunications.Send, "Greeting")]
    public class SendGreetingCommand : Cmdlet
    {
        // Declare the parameters for the cmdlet.
        [Parameter(Mandatory=true)]
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
        private string name;

        // Override the ProcessRecord method to process
        // the supplied user name and write out a
        // greeting to the user by calling the WriteObject
        // method.
        protected override void ProcessRecord()
        {
            WriteObject("Hello " + name + "!");
        }
    }
}
```

See also

[System.Management.Automation.Cmdlet](#)

[System.Management.Automation.PSCmdlet](#)

[System.Management.Automation.Cmdlet.ProcessRecord](#)

[System.Management.Automation.Cmdlet.WriteObject](#)

[CmdletAttribute Declaration](#)

[ParameterAttribute Declaration](#)

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to Declare Cmdlet Parameters

Article • 09/17/2021

These examples show how to declare named, positional, required, optional, and switch parameters. These examples also show how to define a parameter alias.

How to Declare a Named Parameter

- Define a public property as shown in the following code. When you add the `Parameter` attribute, omit the `Position` keyword from the attribute.

```
C#  
  
[Parameter()]
public string UserName
{
    get { return userName; }
    set { userName = value; }
}
private string userName;
```

For more information about the `Parameter` attribute, see [Parameter Attribute Declaration](#).

How to Declare a Positional Parameter

- Define a public property as shown in the following code. When you add the `Parameter` attribute, set the `Position` keyword to the argument position. A value of 0 indicates the first position.

```
C#  
  
[Parameter(Position = 0)]
public string UserName
{
    get { return userName; }
    set { userName = value; }
}
private string userName;
```

For more information about the `Parameter` attribute, see [Parameter Attribute Declaration](#).

How to Declare a Mandatory Parameter

- Define a public property as shown in the following code. When you add the Parameter attribute, set the `Mandatory` keyword to `true`.

C#

```
[Parameter(Position = 0, Mandatory = true)]
public string UserName
{
    get { return userName; }
    set { userName = value; }
}
private string userName;
```

For more information about the Parameter attribute, see [Parameter Attribute Declaration](#).

How to Declare an Optional Parameter

- Define a public property as shown in the following code. When you add the Parameter attribute, omit the `Mandatory` keyword.

C#

```
[Parameter(Position = 0)]
public string UserName
{
    get { return userName; }
    set { userName = value; }
}
private string userName;
```

How to Declare a Switch Parameter

- Define a public property as type `System.Management.Automation.SwitchParameter`, and then declare the Parameter attribute.

C#

```
[Parameter(Position = 1)]
public SwitchParameter GoodBye
{
    get { return goodbye; }
```

```
    set { goodbye = value; }
}
private bool goodbye;
```

For more information about the Parameter attribute, see [Parameter Attribute Declaration](#).

How to Declare a Parameter with Aliases

- Define a public property as shown in the following code. Add an Alias attribute that lists the aliases for the parameter. In this example, three aliases are defined for the same parameter. The first alias provides a shortcut. The second and third aliases provide names you can use for different scenarios.

```
C#  
  
[Alias("UN","Writer","Editor")]
[Parameter()]
public string UserName
{
    get { return userName; }
    set { userName = value; }
}
private string userName;
```

For more information about the Alias attribute, see [Alias Attribute Declaration](#).

See Also

[System.Management.Automation.SwitchParameter](#)

[Parameter Attribute Declaration](#)

[Alias Attribute Declaration](#)

[Writing a Windows PowerShell Cmdlet](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to Declare Parameter Sets

Article • 09/17/2021

This example shows how to define two parameter sets when you declare the parameters for a cmdlet. Each parameter set has both a unique parameter and a shared parameter that is used by both parameter sets. For more information about parameters sets, including how to specify the default parameter set, see [Cmdlet Parameter Sets](#).

ⓘ Important

Whenever possible, define the unique parameter of a parameter set as a required parameter. However, if you want your cmdlet to run without specifying any parameters, the unique parameter can be an optional parameter. For example, the unique parameter of the `Get-Command` cmdlet is optional.

How to Define Two Parameter Sets

1. Add the `ParameterSet` keyword to the `Parameter` attribute for the unique parameter of the first parameter set.

C#

```
[Parameter(Position = 0, Mandatory = true,
          ParameterSetName = "Test01")]
public string UserName
{
    get { return userName; }
    set { userName = value; }
}
private string userName;
```

2. Add the `ParameterSet` keyword to the `Parameter` attribute for the unique parameter of the second parameter set.

C#

```
[Parameter(Position = 0, Mandatory = true,
          ParameterSetName = "Test02")]
public string ComputerName
{
    get { return computerName; }
    set { computerName = value; }
```

```
}
```

```
private string computerName;
```

3. For the parameter that belongs to both parameter sets, add a Parameter attribute for each parameter set and then add the `ParameterSet` keyword to each set. In each Parameter attribute, you can specify how that parameter is defined. A parameter can be optional in one set and mandatory in another.

C#

```
[Parameter(Mandatory= true, ParameterSetName = "Test01")]
[Parameter(ParameterSetName = "Test02")]
public string SharedParam
{
    get { return sharedParam; }
    set { sharedParam = value; }
}
private string sharedParam;
```

See Also

[Cmdlet Parameter Sets](#)

[Writing a Windows PowerShell Cmdlet](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to Validate Parameter Input

Article • 09/17/2021

This section contains examples that show how to validate parameter input by using various attributes to implement validation rules.

In This Section

[How to Validate an Argument with a Script](#) Describes how to validate an argument set by using the `ArgumentSet` attribute.

[How to Validate an Argument Set](#) Describes how to validate an argument set by using the `ArgumentSet` attribute.

[How to Validate an Argument Range](#) Describes how to validate an argument range by using the `ArgumentRange` attribute.

[How to Validate an Argument Pattern](#) Describes how to validate an argument pattern by using the `ArgumentPattern` attribute.

[How to Validate the Argument Length](#) Describes how to validate the length of an argument by using the `ArgumentLength` attribute.

[How to Validate an Argument Count](#) Describes how to validate an argument count by using the `ArgumentCount` attribute.

The way a parameter is declared can affect validation. For more information, see [How to Declare Cmdlet Parameters](#).

Reference

See Also

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to validate an argument using a script

Article • 06/11/2024

This example shows how to specify a validation rule that uses a script to check the parameter argument before the cmdlet is run. The value of the parameter is piped to the script. The script must return `$true` for every value piped to it.

ⓘ Note

For more information about the class that defines this attribute, see [System.Management.Automation.ValidateScriptAttribute](#).

To validate an argument using a script

- Add the `ValidateScript` attribute as shown in the following code. This example specifies a script to validate that the input value is an odd number.

C#

```
[ValidateScript("$_ % 2", ErrorMessage = "The item '{0}' did not pass validation of script '{1}'")]
[Parameter(Position = 0, Mandatory = true)]
public int32 OddNumber
{
    get { return oddNumber; }
    set { oddNumber = value; }
}

private int32 oddNumber;
```

For more information about how to declare this attribute, see [ValidateScript Attribute Declaration](#).

See Also

[System.Management.Automation.ValidateScriptAttribute](#)

[ValidateScript Attribute Declaration](#)

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to Validate an Argument Set

Article • 03/24/2025

This example shows how to specify a validation rule that the Windows PowerShell runtime can use to check the parameter argument before the cmdlet is run. This validation rule provides a set of the valid values for the parameter argument.

ⓘ Note

For more information about the class that defines this attribute, see

[System.Management.Automation.ValidateSetAttribute](#).

To validate an argument set

- Add the ValidateSet attribute as shown in the following code. This example specifies a set of three possible values for the `UserName` parameter.

C#

```
[ValidateSet("Steve", "Mary", "Carl", IgnoreCase = true)]
[Parameter(Position = 0, Mandatory = true)]
public string UserName
{
    get { return userName; }
    set { userName = value; }
}

private string userName;
```

For more information about how to declare this attribute, see [ValidateSet Attribute Declaration](#).

See Also

[System.Management.Automation.ValidateSetAttribute](#)

[ValidateSet Attribute Declaration](#)

[Writing a Windows PowerShell Cmdlet](#)

How to Validate an Argument Range

Article • 03/24/2025

This example shows how to specify a validation rule that the Windows PowerShell runtime can use to check the minimum and maximum values of the parameter argument before the cmdlet is run. You set this validation rule by declaring the `ValidateRange` attribute.

 **Note**

For more information about the class that defines this attribute, see [System.Management.Automation.ValidateRangeAttribute](#).

To validate an argument range

- Add the `ValidateRange` attribute as shown in the following code. This example specifies a range of 0 to 5 for the `InputData` parameter.

C#

```
[ValidateRange(0, 5)]
[Parameter(Position = 0, Mandatory = true)]
public int InputData
{
    get { return inputData; }
    set { inputData = value; }
}
private int inputData;
```

For more information about how to declare this attribute, see [ValidateRange Attribute Declaration](#).

See Also

[ValidateRange Attribute Declaration](#)

[Writing a Windows PowerShell Cmdlet](#)

How to Validate an Argument Pattern

Article • 09/17/2021

This example shows how to specify a validation rule that the Windows PowerShell runtime can use to check the character pattern of the parameter argument before the cmdlet is run. You set this validation rule by declaring the `ValidatePattern` attribute.

ⓘ Note

For more information about the class that defines this attribute, see

[System.Management.Automation.ValidatePatternAttribute](#).

To validate an argument pattern

- Add the `Validate` attribute as shown in the following code. This example specifies a pattern of four digits, where each digit has a value of 0 through 9.

C#

```
[ValidatePattern("[0-9][0-9][0-9][0-9]")]
[Parameter(Position = 0, Mandatory = true)]
public int InputData
{
    get { return inputData; }
    set { inputData = value; }
}

private int inputData;
```

For more information about how to declare this attribute, see [ValidatePattern Attribute Declaration](#).

See Also

[ValidatePattern Attribute Declaration](#)

[Writing a Windows PowerShell Cmdlet](#)

How to Validate the Argument Length

Article • 09/17/2021

This example shows how to specify a validation rule that the Windows PowerShell runtime can use to check the number of characters (the length) of the parameter argument before the cmdlet is run. You set this validation rule by declaring the `ValidateLength` attribute.

ⓘ Note

For more information about the class that defines this attribute, see [System.Management.Automation.ValidateLengthAttribute](#).

To validate the argument length

- Add the `Validate` attribute as shown in the following code. This example specifies that the length of the argument should have a length of 0 to 10 characters.

C#

```
[ValidateLength(0, 10)]
[Parameter(Position = 0, Mandatory = true)]
public string UserName
{
    get { return userName; }
    set { userName = value; }
}
private string userName;
```

For more information about how to declare this attribute, see [ValidateLength Attribute Declaration](#).

See Also

[ValidateLength Attribute Declaration](#)

[Writing a Windows PowerShell Cmdlet](#)

How to Validate an Argument Count

Article • 09/17/2021

This example shows how to specify a validation rule that the Windows PowerShell runtime can use to check the number of arguments (the count) that a parameter accepts before the cmdlet is run. You set this validation rule by declaring the `ValidateCount` attribute.

 **Note**

For more information about the class that defines this attribute, see [System.Management.Automation.ValidateCountAttribute](#).

To validate an argument count

- Add the `Validate` attribute as shown in the following code. This example specifies that the parameter will accept one argument or as many as three arguments.

C#

```
[ValidateCount(1, 3)]
[Parameter(Position = 0, Mandatory = true)]
public string[] UserNames
{
    get { return userNames; }
    set { userNames = value; }
}

private string[] userNames;
```

For more information about how to declare this attribute, see [ValidateCount Attribute Declaration](#).

See Also

[ValidateCount Attribute Declaration](#)

[Writing a Windows PowerShell Cmdlet](#)

How to Declare Dynamic Parameters

Article • 03/24/2025

This example shows how to define dynamic parameters that are added to the cmdlet at runtime. In this example, the `Department` parameter is added to the cmdlet whenever the user specifies the `Employee` switch parameter. For more information about dynamic parameters, see [Cmdlet Dynamic Parameters](#).

To define dynamic parameters

1. In the cmdlet class declaration, add the `System.Management.Automation.IDynamicParameters` interface as shown.

```
C#  
  
public class SendGreetingCommand : Cmdlet, IDynamicParameters
```

2. Call the

`System.Management.Automation.IDynamicParameters.GetDynamicParameters*` method, which returns the object in which the dynamic parameters are defined. In this example, the method is called when the `Employee` parameter is specified.

```
C#  
  
public object GetDynamicParameters()  
{  
    if (employee)  
    {  
        context= new SendGreetingCommandDynamicParameters();  
        return context;  
    }  
    return null;  
}  
private SendGreetingCommandDynamicParameters context;
```

3. Declare a class that defines the dynamic parameters to be added. You can use the attributes that you used to declare the static cmdlet parameters to declare the dynamic parameters.

```
C#  
  
public class SendGreetingCommandDynamicParameters  
{
```

```

[Parameter]
[ValidateSet ("Marketing", "Sales", "Development")]
public string Department
{
    get { return department; }
    set { department = value; }
}
private string department;
}

```

Example

In this example, the `Department` parameter is added whenever the user specifies the `Employee` parameter. The `Department` parameter is an optional parameter, and the `ValidateSet` attribute is used to specify the allowed arguments.

C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Management.Automation;      // PowerShell assembly.

namespace SendGreeting
{
    // Declare the cmdlet class that supports the
    // IDynamicParameters interface.
    [Cmdlet(VerbsCommunications.Send, "Greeting")]
    public class SendGreetingCommand : Cmdlet, IDynamicParameters
    {
        // Declare the parameters for the cmdlet.
        [Parameter(Mandatory = true)]
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
        private string name;

        [Parameter]
        [Alias ("FTE")]
        public SwitchParameter Employee
        {
            get { return employee; }
            set { employee = value; }
        }
        private Boolean employee;

        // Implement GetDynamicParameters to
        // retrieve the dynamic parameter.
    }
}

```

```

public object GetDynamicParameters()
{
    if (employee)
    {
        context= new SendGreetingCommandDynamicParameters();
        return context;
    }
    return null;
}
private SendGreetingCommandDynamicParameters context;

// Override the ProcessRecord method to process the
// supplied user name and write out a greeting to
// the user by calling the WriteObject method.
protected override void ProcessRecord()
{
    WriteObject("Hello " + name + "! ");
    if (employee)
    {
        WriteObject("Department: " + context.Department);
    }
}
}

// Define the dynamic parameters to be added
public class SendGreetingCommandDynamicParameters
{
    [Parameter]
    [ValidateSet ("Marketing", "Sales", "Development")]
    public string Department
    {
        get { return department; }
        set { department = value; }
    }
    private string department;
}
}

```

See Also

- [System.Management.Automation.RuntimeDefinedParameterDictionary](#)
- [System.Management.Automation.IDynamicParameters.GetDynamicParameters*](#)
- [Cmdlet Dynamic Parameters](#)
- [Windows PowerShell SDK](#)

How to Invoke Scripts Within a Cmdlet

Article • 12/05/2023

This example shows how to invoke a script that is supplied to a cmdlet. The script is executed by the cmdlet, and its results are returned to the cmdlet as a collection of [System.Management.Automation.PSObject](#) objects.

To invoke a script block

1. The command verifies that a script block was supplied to the cmdlet. If a script block was supplied, the command invokes the script block with its required parameters.

```
C#  
  
if (script != null)  
{  
    WriteDebug("Executing script block.");  
  
    // Invoke the script block with the required arguments.  
    Collection<PSObject> PSObjects = script.Invoke(  
        line,  
        simpleMatch,  
        caseSensitive  
    );  
    // more code as needed...  
}
```

2. Then, the script iterates through the returned collection of [System.Management.Automation.PSObject](#) objects and perform the necessary operations.

```
C#  
  
foreach (PSObject object in PSObjects)  
{  
    if (LanguagePrimitives.IsTrue(object))  
    {  
        result = new MatchInfo();  
        result.Line = line;  
        result.IgnoreCase = !caseSensitive;  
        break;  
    }  
}
```

See Also

[Writing a Windows PowerShell Cmdlet](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

How to Override Input Processing Methods

Article • 09/17/2021

These examples show how to overwrite the input processing methods within a cmdlet. These methods are used to perform the following operations:

- The [System.Management.Automation.Cmdlet.BeginProcessing](#) method is used to perform one-time startup operations that are valid for all the objects processed by the cmdlet. The Windows PowerShell runtime calls this method only once.
- The [System.Management.Automation.Cmdlet.ProcessRecord](#) method is used to process the objects passed to the cmdlet. The Windows PowerShell runtime calls this method for each object passed to the cmdlet.
- The [System.Management.Automation.Cmdlet.EndProcessing](#) method is used to perform one-time post processing operations. The Windows PowerShell runtime calls this method only once.

To override the BeginProcessing method

- Declare a protected override of the [System.Management.Automation.Cmdlet.BeginProcessing](#) method.

The following class prints a sample message. To use this class, change the verb and noun in the Cmdlet attribute, change the name of the class to reflect the new verb and noun, and then add the functionality you require to the override of the [System.Management.Automation.Cmdlet.BeginProcessing](#) method.

C#

```
[Cmdlet(VerbsDiagnostic.Test, "BeginProcessingClass")]
public class TestBeginProcessingClassTemplate : Cmdlet
{
    // Override the BeginProcessing method to add preprocessing
    //operations to the cmdlet.
    protected override void BeginProcessing()
    {
        // Replace the WriteObject method with the logic required
        // by your cmdlet. It is used here to generate the following
        // output:
        // "This is a test of the BeginProcessing template."
        WriteObject("This is a test of the BeginProcessing template.");
    }
}
```

```
}
```

To override the ProcessRecord method

- Declare a protected override of the [System.Management.Automation.Cmdlet.ProcessRecord](#) method.

The following class prints a sample message. To use this class, change the verb and noun in the Cmdlet attribute, change the name of the class to reflect the new verb and noun, and then add the functionality you require to the override of the [System.Management.Automation.Cmdlet.ProcessRecord](#) method.

C#

```
[Cmdlet(VerbsDiagnostic.Test, "ProcessRecordClass")]
public class TestProcessRecordClassTemplate : Cmdlet
{
    // Override the ProcessRecord method to add processing
    //operations to the cmdlet.
    protected override void ProcessRecord()
    {
        // Replace the WriteObject method with the logic required
        // by your cmdlet. It is used here to generate the following
        // output:
        // "This is a test of the ProcessRecord template."
        WriteObject("This is a test of the ProcessRecord template.");
    }
}
```

To override the EndProcessing method

- Declare a protected override of the [System.Management.Automation.Cmdlet.EndProcessing](#) method.

The following class prints a sample. To use this class, change the verb and noun in the Cmdlet attribute, change the name of the class to reflect the new verb and noun, and then add the functionality you require to the override of the [System.Management.Automation.Cmdlet.EndProcessing](#) method.

C#

```
[Cmdlet(VerbsDiagnostic.Test, "EndProcessingClass")]
public class TestEndProcessingClassTemplate : Cmdlet
```

```
{  
    // Override the EndProcessing method to add postprocessing  
    //operations to the cmdlet.  
    protected override void EndProcessing()  
    {  
        // Replace the WriteObject method with the logic required  
        // by your cmdlet. It is used here to generate the following  
        // output:  
        // "This is a test of the BeginProcessing template."  
        WriteObject("This is a test of the EndProcessing template.");  
    }  
}
```

See Also

[System.Management.Automation.Cmdlet.BeginProcessing](#)

[System.Management.Automation.Cmdlet.EndProcessing](#)

[System.Management.Automation.Cmdlet.ProcessRecord](#)

[Writing a Windows PowerShell Cmdlet](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Requesting Confirmation from Cmdlets

Article • 09/17/2021

Cmdlets should request confirmation when they are about to make a change to the system that is outside of the Windows PowerShell environment. For example, if a cmdlet is about to add a user account or stop a process, the cmdlet should require confirmation from the user before it proceeds. In contrast, if a cmdlet is about to change a Windows PowerShell variable, the cmdlet does not need to require confirmation.

In order to make a confirmation request, the cmdlet must indicate that it supports confirmation requests, and it must call the [System.Management.Automation.Cmdlet.ShouldProcess](#) and [System.Management.Automation.Cmdlet.ShouldContinue](#) (optional) methods to display a confirmation request message.

Supporting Confirmation Requests

To support confirmation requests, the cmdlet must set the `SupportsShouldProcess` parameter of the `Cmdlet` attribute to `true`. This enables the `Confirm` and `WhatIf` cmdlet parameters that are provided by Windows PowerShell. The `Confirm` parameter allows the user to control whether the confirmation request is displayed. The `WhatIf` parameter allows the user to determine whether the cmdlet should display a message or perform its action. Do not manually add the `Confirm` and `WhatIf` parameters to a cmdlet.

The following example shows a `Cmdlet` attribute declaration that supports confirmation requests.

C#

```
[Cmdlet(VerbsDiagnostic.Test, "RequestConfirmationTemplate1",
    SupportsShouldProcess = true)]
```

Calling the Confirmation request methods

In the cmdlet code, call the [System.Management.Automation.Cmdlet.ShouldProcess](#) method before the operation that changes the system is performed. Design the cmdlet so that if the call returns a value of `false`, the operation is not performed, and the cmdlet processes the next operation.

Calling the ShouldContinue Method

Most cmdlets request confirmation using only the [System.Management.Automation.Cmdlet.ShouldProcess](#) method. However, some cases might require additional confirmation. For these cases, supplement the [System.Management.Automation.Cmdlet.ShouldProcess](#) call with a call to the [System.Management.Automation.Cmdlet.ShouldContinue](#) method. This allows the cmdlet or provider to more finely control the scope of the **Yes to all** response to the confirmation prompt.

If a cmdlet calls the [System.Management.Automation.Cmdlet.ShouldContinue](#) method, the cmdlet must also provide a [Force](#) switch parameter. If the user specifies [Force](#) when the user invokes the cmdlet, the cmdlet should still call [System.Management.Automation.Cmdlet.ShouldProcess](#), but it should bypass the call to [System.Management.Automation.Cmdlet.ShouldContinue](#).

[System.Management.Automation.Cmdlet.ShouldContinue](#) will throw an exception when it is called from a non-interactive environment where the user cannot be prompted. Adding a [Force](#) parameter ensures that the command can still be performed when it is invoked in a non-interactive environment.

The following example shows how to call [System.Management.Automation.Cmdlet.ShouldProcess](#) and [System.Management.Automation.Cmdlet.ShouldContinue](#).

```
C#  
  
if (ShouldProcess (...))  
{  
    if (Force || ShouldContinue(...))  
    {  
        // Add code that performs the operation.  
    }  
}
```

The behavior of a [System.Management.Automation.Cmdlet.ShouldProcess](#) call can vary depending on the environment in which the cmdlet is invoked. Using the previous guidelines will help ensure that the cmdlet behaves consistently with other cmdlets, regardless of the host environment.

For an example of calling the [System.Management.Automation.Cmdlet.ShouldProcess](#) method, see [How to Request Confirmations](#).

Specify the Impact Level

When you create the cmdlet, specify the impact level (the severity) of the change. To do this, set the value of the `ConfirmImpact` parameter of the `Cmdlet` attribute to High, Medium, or Low. You can specify a value for `ConfirmImpact` only when you also specify the `SupportsShouldProcess` parameter for the cmdlet.

For most cmdlets, you do not have to explicitly specify `ConfirmImpact`. Instead, use the default setting of the parameter, which is Medium. If you set `ConfirmImpact` to High, the operation will be confirmed by default. Reserve this setting for highly disruptive actions, such as reformatting a hard-disk volume.

Calling Non-Confirmation Methods

If the cmdlet or provider must send a message but not request confirmation, it can call the following three methods. Avoid using the `System.Management.Automation.Cmdlet.WriteObject` method to send messages of these types because `System.Management.Automation.Cmdlet.WriteObject` output is intermingled with the normal output of your cmdlet or provider, which makes script writing difficult.

- To caution the user and continue with the operation, the cmdlet or provider can call the `System.Management.Automation.Cmdlet.WriteWarning` method.
- To provide additional information that the user can retrieve using the `Verbose` parameter, the cmdlet or provider can call the `System.Management.Automation.Cmdlet.WriteVerbose` method.
- To provide debugging-level detail for other developers or for product support, the cmdlet or provider can call the `System.Management.Automation.Cmdlet.WriteDebug` method. The user can retrieve this information using the `Debug` parameter.

Cmdlets and providers first call the following methods to request confirmation before they attempt to perform an operation that changes a system outside of Windows PowerShell:

- `System.Management.Automation.Cmdlet.ShouldProcess`
- `System.Management.Automation.Provider.CmdletProvider.ShouldProcess`

They do so by calling the `System.Management.Automation.Cmdlet.ShouldProcess` method, which prompts the user to confirm the operation based on how the user invoked the command.

See Also

Writing a Windows PowerShell Cmdlet

How to Support Transactions

Article • 09/17/2021

This example shows the basic code elements that add support for transactions to a cmdlet.

ⓘ Important

For more information about how Windows PowerShell handles transactions, see [About Transactions](#).

To support transactions

- When you declare the Cmdlet attribute, specify that the cmdlet supports transactions. When the cmdlet supports transactions, Windows PowerShell adds the `UseTransaction` parameter to the cmdlet when it is run.

C#

```
[Cmdlet(VerbsCommunications.Send, "GreetingTx",
    SupportsTransactions=true )]
```

- Within one of the input processing methods, add an `if` block to determine if a transaction is available. If the `if` statement resolves to `true`, the actions within this statement can be performed within the context of the current transaction.

C#

```
if (TransactionAvailable())
{
    using (CurrentPSTransaction)
    {
        WriteObject("Hello " + name + " from within a transaction.");
    }
}
```

See Also

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to Support Jobs

Article • 09/23/2021

This example shows how to support jobs when you write cmdlets. If you want users to run your cmdlet as a background job, you must include the code described in the following procedure. For more information about background jobs, see [Background Jobs](#).

To support jobs

1. Define an `AsJob` switch parameter so that the user can decide whether to run the cmdlet as a job.

The following example shows an `AsJob` parameter declaration.

```
C#  
  
[Parameter()]
public SwitchParameter AsJob
{
    get { return asjob; }
    set { asjob = value; }
}
private bool asjob;
```

2. Create an object that derives from the `System.Management.Automation.Job` class. This object can be a custom job object or one of the job objects provided by Windows PowerShell, such a `System.Management.Automation.PSEventJob` object.

The following example shows a custom job object.

```
C#  
  
private SampleJob job = new SampleJob("Get-ProcAsJob");
```

3. In a record processing method, add an `if` statement to detect whether the cmdlet should run as a job. The following code uses the `System.Management.Automation.Cmdlet.ProcessRecord` method.

```
C#  
  
protected override void ProcessRecord()
{
```

```

if (asjob)
{
    // Add the job definition to the job repository,
    // return the job object, and then create the thread
    // used to run the job.
    JobRepository.Add(job);
    WriteObject(job);
    ThreadPool.QueueUserWorkItem(WorkItem);
}
else
{
    job.ProcessJob();
    foreach (PSObject p in job.Output)
    {
        WriteObject(p);
    }
}
}

```

4. For custom job objects, implement the job class.

```

C#

private class SampleJob : Job
{
    internal SampleJob(string command)
        : base(command)
    {
        SetJobState(JobState.NotStarted);
    }
    public override string StatusMessage
    {
        get { throw new NotImplementedException(); }
    }

    public override bool HasMoreData
    {
        get
        {
            return hasMoreData;
        }
    }
    private bool hasMoreData = true;

    public override string Location
    {
        get { throw new NotImplementedException(); }
    }

    public override void StopJob()
    {
        throw new NotImplementedException();
    }
}

```

```

internal void ProcessJob()
{
    SetJobState(JobState.Running);
    DoProcessLogic();
    SetJobState(JobState.Completed);
}

// Retrieve the processes of the local computer.
void DoProcessLogic()
{
    Process[] p = Process.GetProcesses();

    foreach (Process pl in p)
    {
        Output.Add(PSObject.AsPSObject(pl));
    }
    Output.Complete();
} // End DoProcessLogic.
} // End SampleJob class.

```

5. If the cmdlet performs the work, call the

`System.Management.Automation.Cmdlet.WriteObject` method to return a process object to the pipeline. If the work is performed as a job, add child job to the job.

C#

```

void DoProcessLogic(bool asJob)
{
    Process[] p = Process.GetProcesses();

    foreach (Process pl in p)
    {
        if (!asJob)
        {
            WriteObject(pl);
        }
        else
        {
            job.ChildJobs[0].Output.Add(PSObject.AsPSObject(pl));
        }
    }
} // End DoProcessLogic.

```

Example

The following sample code shows the code for a `Get-Proc` cmdlet that can retrieve processes internally or by using a background job.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Management.Automation; // Windows PowerShell namespace.
using System.Threading;           // Thread pool namespace for posting
                                // work.
using System.Diagnostics;        // Diagnostics namespace for retrieving
                                // process objects.

// This sample shows a cmdlet whose work can be done by the cmdlet or by
using
// a background job. Background jobs are executed in their own thread,
// independent of the pipeline thread in which the cmdlet is executed.
//
// To load this cmdlet, create a module folder and copy the
GetProcessSample06.dll
// assembly into the module folder. Make sure that the path to the module
folder
// is added to the $PSModulePath environment variable.
// Module folder path:
//      user/documents/WindowsPowerShell/modules/GetProcessSample06
//
// To import the module, run the following command: Import-Module
GetProcessSample06.
// To test the cmdlet, run the following command: Get-Proc -Name <process
name>
//

//
namespace Microsoft.Samples.PowerShell.Commands
{
    /// <summary>
    /// This cmdlet retrieves process internally or returns
    /// a job that retrieves the processes.
    /// </summary>
    [Cmdlet(VerbsCommon.Get, "Proc")]
    public sealed class GetProcCommand : PSCmdlet
    {

        #region Parameters
        /// <summary>
        /// Specify the Name parameter. This parameter accepts
        /// process names from the command line.
        /// </summary>
        [Parameter(
            Position = 0,
            ValueFromPipeline = true,
            ValueFromPipelineByPropertyName = true)]
        [ValidateNotNullOrEmpty]
        public string[] Name
        {
```

```

        get { return processNames; }
        set { processNames = value; }
    }
    private string[] processNames;

    /// <summary>
    /// Specify the AsJob parameter. This parameter indicates
    /// whether the cmdlet should retrieve the processes internally
    /// or return a Job object that retrieves the processes.
    /// </summary>
    [Parameter()]
    public SwitchParameter AsJob
    {
        get { return asjob; }
        set { asjob = value; }
    }
    private bool asjob;

#endregion Parameters

#region Cmdlet Overrides

// Create a custom job object.
private SampleJob job = new SampleJob("Get-ProcAsJob");

    /// <summary>
    /// Determines if the processes should be retrieved
    /// internally or if a Job object should be returned.
    /// </summary>
protected override void ProcessRecord()
{
    if (asjob)
    {
        // Add the job definition to the job repository,
        // return the job object, and then create the thread
        // used to run the job.
        JobRepository.Add(job);
        WriteObject(job);
        ThreadPool.QueueUserWorkItem(WorkItem);
    }
    else
    {
        job.ProcessJob();
        foreach (PSObject p in job.Output)
        {
            WriteObject(p);
        }
    }
}
#endregion Overrides

// Implement a custom job that derives
// from the System.Management.Automation.Job class.
private class SampleJob : Job
{

```

```

internal SampleJob(string command)
    : base(command)
{
    SetJobState(JobState.NotStarted);
}
public override string StatusMessage
{
    get { throw new NotImplementedException(); }
}

public override bool HasMoreData
{
    get
    {
        return hasMoreData;
    }
}
private bool hasMoreData = true;

public override string Location
{
    get { throw new NotImplementedException(); }
}

public override void StopJob()
{
    throw new NotImplementedException();
}

internal void ProcessJob()
{
    SetJobState(JobState.Running);
    DoProcessLogic();
    SetJobState(JobState.Completed);
}

// Retrieve the processes of the local computer.
void DoProcessLogic()
{
    Process[] p = Process.GetProcesses();

    foreach (Process pl in p)
    {
        Output.Add(PSObject.AsPSObject(pl));
    }
    Output.Complete();
} // End DoProcessLogic.
} // End SampleJob class.

void WorkItem(object dummy)
{
    job.ProcessJob();
}

// Display the results of the work. If not a job,

```

```
// process objects are returned. If a job, the
// output is added to the job as a child job.
void DoProcessLogic(bool asJob)
{
    Process[] p = Process.GetProcesses();

    foreach (Process pl in p)
    {
        if (!asJob)
        {
            WriteObject(pl);
        }
        else
        {
            job.ChildJobs[0].Output.Add(PSObject.AsPSObject(pl));
        }
    }
} // End DoProcessLogic.
} //End GetProcCommand
}
```

How to Invoke a Cmdlet from Within a Cmdlet

Article • 10/20/2022

This example shows how to invoke a binary cmdlet that derives from `[System.Management.Automation.Cmdlet]` directly from within another binary cmdlet, which allows you to add the functionality of the invoked cmdlet to the binary cmdlet you are developing. In this example, the `Get-Process` cmdlet is invoked to get the processes that are running on the local computer. The call to the `Get-Process` cmdlet is equivalent to the following command. This command retrieves all the processes whose names start with the characters "a" through "t".

PowerShell

```
Get-Process -Name [a-t]*
```

ⓘ Important

You can invoke only those cmdlets that derive directly from the `System.Management.Automation.Cmdlet` class. You can't invoke a cmdlet that derives from the `System.Management.Automation.PSCmdlet` class. For an example, see [How to invoke a PSCmdlet from within a PSCmdlet](#).

To invoke a cmdlet from within a cmdlet

1. Ensure that the assembly that defines the cmdlet to be invoked is referenced and that the appropriate `using` statement is added. In this example, the following namespaces are added.

C#

```
using System.Diagnostics;
using System.Management.Automation; // PowerShell assembly.
using Microsoft.PowerShell.Commands; // PowerShell cmdlets assembly
you want to call.
```

2. In the input processing method of the cmdlet, create a new instance of the cmdlet to be invoked. In this example, an object of type

`Microsoft.PowerShell.Commands.GetProcessCommand` is created along with the string that contains the arguments that are used when the cmdlet is invoked.

```
C#
```

```
GetProcessCommand gp = new GetProcessCommand();
gp.Name = new string[] { "[a-t]*" };
```

3. Call the `System.Management.Automation.Cmdlet.Invoke*` method to invoke the `Get-Process` cmdlet.

```
C#
```

```
foreach (Process p in gp.Invoke<Process>())
{
    Console.WriteLine(p.ToString());
}
```

Example

In this example, the `Get-Process` cmdlet is invoked from within the `System.Management.Automation.Cmdlet.BeginProcessing` method of a cmdlet.

```
C#
```

```
using System;
using System.Diagnostics;
using System.Management.Automation; // PowerShell assembly.
using Microsoft.PowerShell.Commands; // PowerShell cmdlets assembly you
want to call.

namespace SendGreeting
{
    // Declare the class as a cmdlet and specify an
    // appropriate verb and noun for the cmdlet name.
    [Cmdlet(VerbsCommunications.Send, "GreetingInvoke")]
    public class SendGreetingInvokeCommand : Cmdlet
    {
        // Declare the parameters for the cmdlet.
        [Parameter(Mandatory = true)]
        public string Name { get; set; }

        // Override the BeginProcessing method to invoke
        // the Get-Process cmdlet.
        protected override void BeginProcessing()
        {
            GetProcessCommand gp = new GetProcessCommand();
```

```
gp.Name = new string[] { "[a-t]*" };
foreach (Process p in gp.Invoke<Process>())
{
    WriteVerbose(p.ToString());
}
}

// Override the ProcessRecord method to process
// the supplied user name and write out a
// greeting to the user by calling the WriteObject
// method.
protected override void ProcessRecord()
{
    WriteObject("Hello " + Name + "!");
}
}
```

See Also

[Writing a Windows PowerShell Cmdlet](#)

How to invoke a PSCmdlet from within a PSCmdlet

Article • 10/20/2022

This example shows how to invoke a script based cmdlet or binary cmdlet inheriting from `[System.Management.Automation.PSCmdlet]` from within a binary cmdlet. In this example, the new cmdlet `Get-ClipboardReverse` calls `Get-Clipboard` to get the contents of the clipboard. The `Get-ClipboardReverse` reverses the order of the characters and returns the reversed string.

ⓘ Note

The `[PSCmdlet]` class differs from the `[Cmdlet]` class. `[PSCmdlet]` implementations use runspace context information so you must invoke another cmdlet using the PowerShell pipeline API. In `[Cmdlet]` implementations you can call the cmdlet's .NET API directly. For an example, see [How to invoke a Cmdlet from within a Cmdlet](#).

To invoke a cmdlet from within a PSCmdlet

1. Ensure that the namespace for the `[System.Management.Automation.PowerShell]` API is referenced. In this example, the following namespaces are added.

C#

```
using System.Management.Automation; // PowerShell assembly.  
using System.Text;
```

2. To invoke a command from within another binary cmdlet you must use the `[PowerShell]` API to construct a new pipeline and add the cmdlet to be invoked. Call the `System.Management.Automation.PowerShell.Invoke<T>()` method to invoke the pipeline.

C#

```
using var ps = PowerShell.Create(RunspaceMode.CurrentRunspace);  
ps.AddCommand("Get-Clipboard").AddParameter("Raw");  
var output = ps.Invoke<string>();
```

Example

To invoke a script based cmdlet or binary cmdlet inheriting from `[PSCmdlet]` you must build a PowerShell pipeline with the command and parameters you want to execute, then invoke the pipeline.

C#

```
using System;
using System.Management.Automation; // PowerShell assembly.
using System.Text;

namespace ClipboardReverse
{
    [Cmdlet(VerbsCommon.Get, "ClipboardReverse")]
    [OutputType(typeof(string))]
    public class ClipboardReverse : PSCmdlet
    {
        protected override void EndProcessing()
        {
            using var ps = PowerShell.Create(RunspaceMode.CurrentRunspace);
            ps.AddCommand("Get-Clipboard").AddParameter("Raw");
            var output = ps.Invoke<string>();
            if (ps.HadErrors)
            {
                WriteError(new ErrorRecord(ps.Streams.Error[0].Exception,
                    "Get-Clipboard Error",
                    ErrorCategory.NotSpecified, null));
            }
            else
            {
                var sb = new StringBuilder();
                foreach (var text in output)
                {
                    sb.Append(text);
                }

                var reversed = sb.ToString().ToCharArray();
                Array.Reverse(reversed);
                WriteObject(new string(reversed));
            }
        }
    }
}
```

See Also

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorials for Writing Cmdlets

Article • 09/17/2021

This section contains tutorials for writing cmdlets. These tutorials include the code needed to write the cmdlets, plus an explanation of why the code is needed. These topics will be very helpful for those who are just starting to write cmdlets.

Important

For those who want code examples with less description, see [Cmdlet Samples](#).

In This Section

[GetProc Tutorial](#) - This tutorial describes how to define a cmdlet class and add basic functionality such as adding parameters and reporting errors. The cmdlet described in this tutorial is very similar to the [Get-Process](#) cmdlet provided by Windows PowerShell.

[StopProc Tutorial](#) - This tutorial describes how to define a cmdlet and add functionality such as user prompts, wildcard support, and the use of parameter sets. The cmdlet described here performs the same task as the [Stop-Process](#) cmdlet provided by Windows PowerShell.

[SelectStr Tutorial](#) - This tutorial describes how to define a cmdlet that accesses a data store. The cmdlet described here performs the same task as the [Select-String](#) cmdlet provided by Windows PowerShell.

See Also

[GetProc Tutorial](#)

[StopProc Tutorial](#)

[SelectStr Tutorial](#)

[Windows PowerShell SDK](#)



PowerShell feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

GetProc Tutorial

Article • 09/15/2023

This section provides a tutorial for creating a Get-Proc cmdlet that is very similar to the [Get-Process](#) cmdlet provided by Windows PowerShell. This tutorial provides fragments of code that illustrate how cmdlets are implemented, and an explanation of the code.

Topics in this Tutorial

The topics in this tutorial are designed to be read sequentially, with each topic building on what was discussed in the previous topic.

- [Creating a Cmdlet without Parameters](#): This section describes how to create a cmdlet that retrieves information from the local computer without the use of parameters, and then writes the information to the pipeline.
- [Adding Parameters that Process Command-Line Input](#): This section describes how to add a parameter to the Get-Proc cmdlet so that the cmdlet can process input based on explicit objects passed to the cmdlet. The implementation described here retrieves processes based on their name, and then writes the information to the pipeline.
- [Adding Parameters that Process Pipeline Input](#): This section describes how to add a parameter to the Get-Proc cmdlet so that the cmdlet can process objects passed to it through the pipeline. The implementation cmdlet described here retrieves processes based on objects passed to the cmdlet, and then writes the information to the pipeline.
- [Adding Non-terminating Error Reporting to Your Cmdlet](#): This section describes how to add non-terminating error reporting to a cmdlet. The implementation described here detects non-terminating errors that occur when processing input, and writes an error record to the error stream.

See Also

- [Windows PowerShell SDK](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Creating a Cmdlet without Parameters

Article • 09/17/2021

This section describes how to create a cmdlet that retrieves information from the local computer without the use of parameters, and then writes the information to the pipeline. The cmdlet described here is a Get-Proc cmdlet that retrieves information about the processes of the local computer, and then displays that information at the command line.

ⓘ Note

Be aware that when writing cmdlets, the Windows PowerShell® reference assemblies are downloaded onto disk (by default at C:\Program Files\Reference Assemblies\Microsoft\WindowsPowerShell\v1.0). They are not installed in the Global Assembly Cache (GAC).

Naming the Cmdlet

A cmdlet name consists of a verb that indicates the action the cmdlet takes and a noun that indicates the items that the cmdlet acts upon. Because this sample Get-Proc cmdlet retrieves process objects, it uses the verb "Get", defined by the [System.Management.Automation.VerbsCommon](#) enumeration, and the noun "Proc" to indicate that the cmdlet works on process items.

When naming cmdlets, do not use any of the following characters: # , () {} [] & - ^ \$; : " ' < > | ? @ ` .

Choosing a Noun

You should choose a noun that is specific. It is best to use a singular noun prefixed with a shortened version of the product name. An example cmdlet name of this type is "Get-SQLServer".

Choosing a Verb

You should use a verb from the set of approved cmdlet verb names. For more information about the approved cmdlet verbs, see [Cmdlet Verb Names](#).

Defining the Cmdlet Class

Once you have chosen a cmdlet name, define a .NET class to implement the cmdlet. Here is the class definition for this sample Get-Proc cmdlet:

C#

```
[Cmdlet(VerbsCommon.Get, "Proc")]
public class GetProcCommand : Cmdlet
```

VB

```
<Cmdlet(VerbsCommon.Get, "Proc")> _
Public Class GetProcCommand
    Inherits Cmdlet
```

Notice that previous to the class definition, the [System.Management.Automation.CmdletAttribute](#) attribute, with the syntax `[Cmdlet(verb, noun, ...)]`, is used to identify this class as a cmdlet. This is the only required attribute for all cmdlets, and it allows the Windows PowerShell runtime to call them correctly. You can set attribute keywords to further declare the class if necessary. Be aware that the attribute declaration for our sample GetProcCommand class declares only the noun and verb names for the Get-Proc cmdlet.

ⓘ Note

For all Windows PowerShell attribute classes, the keywords that you can set correspond to properties of the attribute class.

When naming the class of the cmdlet, it is a good practice to reflect the cmdlet name in the class name. To do this, use the form "VerbNounCommand" and replace "Verb" and "Noun" with the verb and noun used in the cmdlet name. As is shown in the previous class definition, the sample Get-Proc cmdlet defines a class called GetProcCommand, which derives from the [System.Management.Automation.Cmdlet](#) base class.

ⓘ Important

If you want to define a cmdlet that accesses the Windows PowerShell runtime directly, your .NET class should derive from the [System.Management.Automation.PSCmdlet](#) base class. For more information about this class, see [Creating a Cmdlet that Defines Parameter Sets](#).

Note

The class for a cmdlet must be explicitly marked as public. Classes that are not marked as public will default to internal and will not be found by the Windows PowerShell runtime.

Windows PowerShell uses the [Microsoft.PowerShell.Commands](#) namespace for its cmdlet classes. It is recommended to place your cmdlet classes in a Commands namespace of your API namespace, for example, xxx.PS.Commands.

Overriding an Input Processing Method

The [System.Management.Automation.Cmdlet](#) class provides three main input processing methods, at least one of which your cmdlet must override. For more information about how Windows PowerShell processes records, see [How Windows PowerShell Works](#).

For all types of input, the Windows PowerShell runtime calls [System.Management.Automation.Cmdlet.BeginProcessing](#) to enable processing. If your cmdlet must perform some preprocessing or setup, it can do this by overriding this method.

Note

Windows PowerShell uses the term "record" to describe the set of parameter values supplied when a cmdlet is called.

If your cmdlet accepts pipeline input, it must override the [System.Management.Automation.Cmdlet.ProcessRecord](#) method, and optionally the [System.Management.Automation.Cmdlet.EndProcessing](#) method. For example, a cmdlet might override both methods if it gathers all input using [System.Management.Automation.Cmdlet.ProcessRecord](#) and then operates on the input as a whole rather than one element at a time, as the `Sort-Object` cmdlet does.

If your cmdlet does not take pipeline input, it should override the [System.Management.Automation.Cmdlet.EndProcessing](#) method. Be aware that this method is frequently used in place of [System.Management.Automation.Cmdlet.BeginProcessing](#) when the cmdlet cannot operate on one element at a time, as is the case for a sorting cmdlet.

Because this sample Get-Proc cmdlet must receive pipeline input, it overrides the `System.Management.Automation.Cmdlet.ProcessRecord` method and uses the default implementations for `System.Management.Automation.Cmdlet.BeginProcessing` and `System.Management.Automation.Cmdlet.EndProcessing`. The `System.Management.Automation.Cmdlet.ProcessRecord` override retrieves processes and writes them to the command line using the `System.Management.Automation.Cmdlet.WriteObject` method.

C#

```
protected override void ProcessRecord()
{
    // Get the current processes
    Process[] processes = Process.GetProcesses();

    // Write the processes to the pipeline making them available
    // to the next cmdlet. The second parameter of this call tells
    // PowerShell to enumerate the array, and send one process at a
    // time to the pipeline.
    WriteObject(processes, true);
}
```

VB

```
Protected Overrides Sub ProcessRecord()

    ' Get the current processes.
    Dim processes As Process()
    processes = Process.GetProcesses()

    ' Write the processes to the pipeline making them available
    ' to the next cmdlet. The second parameter of this call tells
    ' PowerShell to enumerate the array, and send one process at a
    ' time to the pipeline.
    WriteObject(processes, True)

End Sub 'ProcessRecord
```

Things to Remember About Input Processing

- The default source for input is an explicit object (for example, a string) provided by the user on the command line. For more information, see [Creating a Cmdlet to Process Command Line Input](#).
- An input processing method can also receive input from the output object of an upstream cmdlet on the pipeline. For more information, see [Creating a Cmdlet to](#)

Process Pipeline Input. Be aware that your cmdlet can receive input from a combination of command-line and pipeline sources.

- The downstream cmdlet might not return for a long time, or not at all. For that reason, the input processing method in your cmdlet should not hold locks during calls to [System.Management.Automation.Cmdlet.WriteObject](#), especially locks for which the scope extends beyond the cmdlet instance.

ⓘ Important

Cmdlets should never call [System.Console.WriteLine*](#) or its equivalent.

- Your cmdlet might have object variables to clean up when it is finished processing (for example, if it opens a file handle in the [System.Management.Automation.Cmdlet.BeginProcessing](#) method and keeps the handle open for use by [System.Management.Automation.Cmdlet.ProcessRecord](#)). It is important to remember that the Windows PowerShell runtime does not always call the [System.Management.Automation.Cmdlet.EndProcessing](#) method, which should perform object cleanup.

For example, [System.Management.Automation.Cmdlet.EndProcessing](#) might not be called if the cmdlet is canceled midway or if a terminating error occurs in any part of the cmdlet. Therefore, a cmdlet that requires object cleanup should implement the complete [System.IDisposable](#) pattern, including the finalizer, so that the runtime can call both [System.Management.Automation.Cmdlet.EndProcessing](#) and [System.IDisposable.Dispose*](#) at the end of processing.

Code Sample

For the complete C# sample code, see [GetProcessSample01 Sample](#).

Defining Object Types and Formatting

Windows PowerShell passes information between cmdlets using .NET objects. Consequently, a cmdlet might need to define its own type, or the cmdlet might need to extend an existing type provided by another cmdlet. For more information about defining new types or extending existing types, see [Extending Object Types and Formatting](#).

Building the Cmdlet

After implementing a cmdlet, you must register it with Windows PowerShell through a Windows PowerShell snap-in. For more information about registering cmdlets, see [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Cmdlet

When your cmdlet has been registered with Windows PowerShell, you can test it by running it on the command line. The code for our sample Get-Proc cmdlet is small, but it still uses the Windows PowerShell runtime and an existing .NET object, which is enough to make it useful. Let's test it to better understand what Get-Proc can do and how its output can be used. For more information about using cmdlets from the command line, see the [Getting Started with Windows PowerShell](#).

1. Start Windows PowerShell, and get the current processes running on the computer.

```
PowerShell
Get-Proc
```

The following output appears.

Output							
Handles	NPM(K)	PM(K)	WS(K)	VS(M)	CPU(s)	Id	ProcessName
254	7	7664	12048	66	173.75	1200	QCTRAY
32	2	1372	2628	31	0.04	1860	DLG
271	6	1216	3688	33	0.03	3816	lg
27	2	560	1920	24	0.01	1768	TpScrex
...							

2. Assign a variable to the cmdlet results for easier manipulation.

```
PowerShell
$p=Get-Proc
```

3. Get the number of processes.

```
PowerShell
$p.Length
```

The following output appears.

```
Output
```

```
63
```

4. Retrieve a specific process.

```
PowerShell
```

```
$p[6]
```

The following output appears.

```
Output
```

Handles	NPM(K)	PM(K)	WS(K)	VS(M)	CPU(s)	Id	ProcessName
1033	3	2400	3336	35	0.53	1588	rundll32

5. Get the start time of this process.

```
PowerShell
```

```
$p[6].StartTime
```

The following output appears.

```
Output
```

```
Tuesday, July 26, 2005 9:34:15 AM
```

```
PowerShell
```

```
$p[6].StartTime.DayOfYear
```

```
Output
```

```
207
```

6. Get the processes for which the handle count is greater than 500, and sort the result.

PowerShell

```
$p | Where-Object {$_.HandleCount -gt 500} | Sort-Object HandleCount
```

The following output appears.

Output

Handles	NPM(K)	PM(K)	WS(K)	VS(M)	CPU(s)	Id	ProcessName
568	14	2164	4972	39	5.55	824	svchost
716	7	2080	5332	28	25.38	468	csrss
761	21	33060	56608	440	393.56	3300	WINWORD
791	71	7412	4540	59	3.31	492	winlogon
...							

7. Use the `Get-Member` cmdlet to list the properties available for each process.

PowerShell

```
$p | Get-Member -MemberType Property
```

Output

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

The following output appears.

Output

Name	MemberType	Definition
BasePriority	Property	System.Int32 BasePriority {get;}
Container	Property	System.ComponentModel.IContainer
Conta...		
EnableRaisingEvents	Property	System.Boolean EnableRaisingEvents
{ge...		
...		

See Also

[Creating a Cmdlet to Process Command Line Input](#)

[Creating a Cmdlet to Process Pipeline Input](#)

[How to Create a Windows PowerShell Cmdlet](#)

[Extending Object Types and Formatting ↗](#)

[How Windows PowerShell Works ↗](#)

[How to Register Cmdlets, Providers, and Host Applications ↗](#)

[Windows PowerShell Reference](#)

[Cmdlet Samples](#)

Adding Parameters That Process Command-Line Input

Article • 09/17/2021

One source of input for a cmdlet is the command line. This topic describes how to add a parameter to the `Get-Proc` cmdlet (which is described in [Creating Your First Cmdlet](#)) so that the cmdlet can process input from the local computer based on explicit objects passed to the cmdlet. The `Get-Proc` cmdlet described here retrieves processes based on their names, and then displays information about the processes at a command prompt.

Defining the Cmdlet Class

The first step in cmdlet creation is cmdlet naming and the declaration of the .NET Framework class that implements the cmdlet. This cmdlet retrieves process information, so the verb name chosen here is "Get." (Almost any sort of cmdlet that is capable of retrieving information can process command-line input.) For more information about approved cmdlet verbs, see [Cmdlet Verb Names](#).

Here's the class declaration for the `Get-Proc` cmdlet. Details about this definition are provided in [Creating Your First Cmdlet](#).

C#

```
[Cmdlet(VerbsCommon.Get, "proc")]
public class GetProcCommand: Cmdlet
```

VB

```
<Cmdlet(VerbsCommon.Get, "Proc")> _
Public Class GetProcCommand
    Inherits Cmdlet
```

Declaring Parameters

A cmdlet parameter enables the user to provide input to the cmdlet. In the following example, `Get-Proc` and `Get-Member` are the names of pipelined cmdlets, and `MemberType` is a parameter for the `Get-Member` cmdlet. The parameter has the argument "property."

PS> `Get-Proc ; Get-Member -MemberType Property`

To declare parameters for a cmdlet, you must first define the properties that represent the parameters. In the `Get-Proc` cmdlet, the only parameter is `Name`, which in this case represents the name of the .NET Framework process object to retrieve. Therefore, the cmdlet class defines a property of type string to accept an array of names.

Here's the parameter declaration for the `Name` parameter of the `Get-Proc` cmdlet.

C#

```
/// <summary>
/// Specify the cmdlet Name parameter.
/// </summary>
[Parameter(Position = 0)]
[ValidateNotNullOrEmpty]
public string[] Name
{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;

#endregion Parameters
```

VB

```
<Parameter(Position:=0), ValidateNotNullOrEmpty()> _
Public Property Name() As String()
    Get
        Return processNames
    End Get

    Set(ByVal value As String())
        processNames = value
    End Set

End Property
```

To inform the Windows PowerShell runtime that this property is the `Name` parameter, a `System.Management.Automation.ParameterAttribute` attribute is added to the property definition. The basic syntax for declaring this attribute is `[Parameter()]`.

ⓘ Note

A parameter must be explicitly marked as public. Parameters that are not marked as public default to internal and are not found by the Windows PowerShell runtime.

This cmdlet uses an array of strings for the `Name` parameter. If possible, your cmdlet should also define a parameter as an array, because this allows the cmdlet to accept more than one item.

Things to Remember About Parameter Definitions

- Predefined Windows PowerShell parameter names and data types should be reused as much as possible to ensure that your cmdlet is compatible with Windows PowerShell cmdlets. For example, if all cmdlets use the predefined `Id` parameter name to identify a resource, user will easily understand the meaning of the parameter, regardless of what cmdlet they are using. Basically, parameter names follow the same rules as those used for variable names in the common language runtime (CLR). For more information about parameter naming, see [Cmdlet Parameter Names](#).
- Windows PowerShell reserves a few parameter names to provide a consistent user experience. Do not use these parameter names: `WhatIf`, `Confirm`, `Verbose`, `Debug`, `Warn`, `ErrorAction`, `ErrorVariable`, `OutVariable`, and `OutBuffer`. Additionally, the following aliases for these parameter names are reserved: `vb`, `db`, `ea`, `ev`, `ov`, and `ob`.
- `Name` is a simple and common parameter name, recommended for use in your cmdlets. It is better to choose a parameter name like this than a complex name that is unique to a specific cmdlet and hard to remember.
- Parameters are case-insensitive in Windows PowerShell, although by default the shell preserves case. Case-sensitivity of the arguments depends on the operation of the cmdlet. Arguments are passed to a parameter as specified at the command line.
- For examples of other parameter declarations, see [Cmdlet Parameters](#).

Declaring Parameters as Positional or Named

A cmdlet must set each parameter as either a positional or named parameter. Both kinds of parameters accept single arguments, multiple arguments separated by commas, and Boolean settings. A Boolean parameter, also called a *switch*, handles only Boolean settings. The switch is used to determine the presence of the parameter. The recommended default is `false`.

The sample `Get-Proc` cmdlet defines the `Name` parameter as a positional parameter with position 0. This means that the first argument the user enters on the command line is automatically inserted for this parameter. If you want to define a named parameter, for which the user must specify the parameter name from the command line, leave the `Position` keyword out of the attribute declaration.

 **Note**

Unless parameters must be named, we recommend that you make the most-used parameters positional so that users will not have to type the parameter name.

Declaring Parameters as Mandatory or Optional

A cmdlet must set each parameter as either an optional or a mandatory parameter. In the sample `Get-Proc` cmdlet, the `Name` parameter is defined as optional because the `Mandatory` keyword is not set in the attribute declaration.

Supporting Parameter Validation

The sample `Get-Proc` cmdlet adds an input validation attribute, `System.Management.Automation.ValidateNotNullOrEmptyAttribute`, to the `Name` parameter to enable validation that the input is neither `null` nor empty. This attribute is one of several validation attributes provided by Windows PowerShell. For examples of other validation attributes, see [Validating Parameter Input](#).

```
[Parameter(Position = 0)]
[ValidateNotNullOrEmpty]
public string[] Name
```

Overriding an Input Processing Method

If your cmdlet is to handle command-line input, it must override the appropriate input processing methods. The basic input processing methods are introduced in [Creating Your First Cmdlet](#).

The `Get-Proc` cmdlet overrides the `System.Management.Automation.Cmdlet.ProcessRecord` method to handle the `Name` parameter input provided by the user or a script. This method gets the processes for each requested process name, or all for processes if no name is provided. Notice that in `System.Management.Automation.Cmdlet.ProcessRecord`, the call to `System.Management.Automation.Cmdlet.WriteObject` is the output mechanism for sending output objects to the pipeline. The second parameter of this call, `enumerateCollection`, is set to `true` to inform the Windows PowerShell runtime to enumerate the output array of process objects and write one process at a time to the command line.

C#

```
protected override void ProcessRecord()
{
    // If no process names are passed to the cmdlet, get all processes.
    if (processNames == null)
    {
        // Write the processes to the pipeline making them available
        // to the next cmdlet. The second argument of this call tells
        // PowerShell to enumerate the array, and send one process at a
        // time to the pipeline.
        WriteObject(Process.GetProcesses(), true);
    }
    else
    {
        // If process names are passed to the cmdlet, get and write
        // the associated processes.
        foreach (string name in processNames)
        {
            WriteObject(Process.GetProcessesByName(name), true);
        }
    }
}
```

VB

```
Protected Overrides Sub ProcessRecord()

    ' If no process names are passed to the cmdlet, get all processes.
    If processNames Is Nothing Then
        Dim processes As Process()
        processes = Process.GetProcesses()
    End If

    ' If process names are specified, write the processes to the
    ' pipeline to display them or make them available to the next cmdlet.

    For Each name As String In processNames
```

```
'/ The second parameter of this call tells PowerShell to enumerate  
the  
'/ array, and send one process at a time to the pipeline.  
WriteObject(Process.GetProcessesByName(name), $True)  
Next  
  
End Sub 'ProcessRecord
```

Code Sample

For the complete C# sample code, see [GetProcessSample02 Sample](#).

Defining Object Types and Formatting

Windows PowerShell passes information between cmdlets by using .NET Framework objects. Consequently, a cmdlet might need to define its own type, or a cmdlet might need to extend an existing type provided by another cmdlet. For more information about defining new types or extending existing types, see [Extending Object Types and Formatting](#).

Building the Cmdlet

After you implement a cmdlet, you must register it with Windows PowerShell by using a Windows PowerShell snap-in. For more information about registering cmdlets, see [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Cmdlet

When your cmdlet is registered with Windows PowerShell, you can test it by running it on the command line. Here are two ways to test the code for the sample cmdlet. For more information about using cmdlets from the command line, see [Getting Started with Windows PowerShell](#).

- At the Windows PowerShell prompt, use the following command to list the Internet Explorer process, which is named "IEXPLORE."

```
PowerShell
```

```
Get-Proc -Name iexplore
```

The following output appears.

Output

Handles	NPM(K)	PM(K)	WS(K)	VS(M)	CPU(s)	Id	ProcessName
354	11	10036	18992	85	0.67	3284	iexplore

- To list the Internet Explorer, Outlook, and Notepad processes named "IEXPLORE," "OUTLOOK," and "NOTEPAD," use the following command. If there are multiple processes, all of them are displayed.

PowerShell

```
Get-Proc -Name iexplore, outlook, notepad
```

The following output appears.

Handles	NPM(K)	PM(K)	WS(K)	VS(M)	CPU(s)	Id	ProcessName
732	21	24696	5000	138	2.25	2288	iexplore
715	19	20556	14116	136	1.78	3860	iexplore
3917	62	74096	58112	468	191.56	1848	OUTLOOK
39	2	1024	3280	30	0.09	1444	notepad
39	2	1024	356	30	0.08	3396	notepad

See Also

[Adding Parameters that Process Pipeline Input](#)

[Creating Your First Cmdlet](#)

[Extending Object Types and Formatting](#)

[How to Register Cmdlets, Providers, and Host Applications](#)

[Windows PowerShell Reference](#)

[Cmdlet Samples](#)

Adding Parameters that Process Pipeline Input

Article • 09/17/2021

One source of input for a cmdlet is an object on the pipeline that originates from an upstream cmdlet. This section describes how to add a parameter to the Get-Proc cmdlet (described in [Creating Your First Cmdlet](#)) so that the cmdlet can process pipeline objects.

This Get-Proc cmdlet uses a `Name` parameter that accepts input from a pipeline object, retrieves process information from the local computer based on the supplied names, and then displays information about the processes at the command line.

Defining the Cmdlet Class

The first step in cmdlet creation is always naming the cmdlet and declaring the .NET class that implements the cmdlet. This cmdlet retrieves process information, so the verb name chosen here is "Get". (Almost any sort of cmdlet that is capable of retrieving information can process command-line input.) For more information about approved cmdlet verbs, see [Cmdlet Verb Names](#).

The following is the definition for this Get-Proc cmdlet. Details of this definition are given in [Creating Your First Cmdlet](#).

C#

```
[Cmdlet(VerbsCommon.Get, "proc")]
public class GetProcCommand : Cmdlet
```

VB

```
<Cmdlet(VerbsCommon.Get, "Proc")> _
Public Class GetProcCommand
    Inherits Cmdlet
```

Defining Input from the Pipeline

This section describes how to define input from the pipeline for a cmdlet. This Get-Proc cmdlet defines a property that represents the `Name` parameter as described in [Adding Parameters that Process Command Line Input](#). (See that topic for general information about declaring parameters.)

However, when a cmdlet needs to process pipeline input, it must have its parameters bound to input values by the Windows PowerShell runtime. To do this, you must add the `ValueFromPipeline` keyword or add the `ValueFromPipelineByPropertyName` keyword to the `System.Management.Automation.ParameterAttribute` attribute declaration. Specify the `ValueFromPipeline` keyword if the cmdlet accesses the complete input object. Specify the `ValueFromPipelineByPropertyName` if the cmdlet accesses only a property of the object.

Here is the parameter declaration for the `Name` parameter of this Get-Proc cmdlet that accepts pipeline input.

C#

```
[Parameter(
    Position = 0,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true)]
[ValidateNotNullOrEmpty]
public string[] Name
{
    get { return this.processNames; }
    set { this.processNames = value; }
}
```

VB

```
<Parameter(Position:=0, ValueFromPipeline:=True, _
ValueFromPipelineByPropertyName:=True), ValidateNotNullOrEmpty()> _
Public Property Name() As String()
    Get
        Return processNames
    End Get

    Set(ByVal value As String())
        processNames = value
    End Set

End Property
```

The previous declaration sets the `ValueFromPipeline` keyword to `true` so that the Windows PowerShell runtime will bind the parameter to the incoming object if the object is the same type as the parameter, or if it can be coerced to the same type. The `ValueFromPipelineByPropertyName` keyword is also set to `true` so that the Windows PowerShell runtime will check the incoming object for a `Name` property. If the incoming object has such a property, the runtime will bind the `Name` parameter to the `Name` property of the incoming object.

ⓘ Note

The setting of the `ValueFromPipeline` attribute keyword for a parameter takes precedence over the setting for the `ValueFromPipelineByPropertyName` keyword.

Overriding an Input Processing Method

If your cmdlet is to handle pipeline input, it needs to override the appropriate input processing methods. The basic input processing methods are introduced in [Creating Your First Cmdlet](#).

This Get-Proc cmdlet overrides the

`System.Management.Automation.Cmdlet.ProcessRecord` method to handle the `Name` parameter input provided by the user or a script. This method will get the processes for each requested process name or all processes if no name is provided. Notice that within `System.Management.Automation.Cmdlet.ProcessRecord`, the call to `WriteObject(System.Object, System.Boolean)` is the output mechanism for sending output objects to the pipeline. The second parameter of this call, `enumerateCollection`, is set to `true` to tell the Windows PowerShell runtime to enumerate the array of process objects, and write one process at a time to the command line.

C#

```
protected override void ProcessRecord()
{
    // If no process names are passed to the cmdlet, get all processes.
    if (processNames == null)
    {
        // Write the processes to the pipeline making them available
        // to the next cmdlet. The second argument of this call tells
        // PowerShell to enumerate the array, and send one process at a
        // time to the pipeline.
        WriteObject(Process.GetProcesses(), true);
    }
    else
    {
        // If process names are passed to the cmdlet, get and write
        // the associated processes.
        foreach (string name in processNames)
        {
            WriteObject(Process.GetProcessesByName(name), true);
        } // End foreach (string name...).
    }
}
```

VB

```
Protected Overrides Sub ProcessRecord()
    Dim processes As Process()

    '/ If no process names are passed to the cmdlet, get all processes.
    If processNames Is Nothing Then
        processes = Process.GetProcesses()
    Else

        '/ If process names are specified, write the processes to the
        '/ pipeline to display them or make them available to the next
        cmdlet.
        For Each name As String In processNames
            '/ The second parameter of this call tells PowerShell to
            enumerate the
            '/ array, and send one process at a time to the pipeline.
            WriteObject(Process.GetProcessesByName(name), True)
        Next
    End If

End Sub 'ProcessRecord
```

Code Sample

For the complete C# sample code, see [GetProcessSample03 Sample](#).

Defining Object Types and Formatting

Windows PowerShell passes information between cmdlets using .NET objects. Consequently, a cmdlet may need to define its own type, or the cmdlet may need to extend an existing type provided by another cmdlet. For more information about defining new types or extending existing types, see [Extending Object Types and Formatting](#).

Building the Cmdlet

After implementing a cmdlet it must be registered with Windows PowerShell through a Windows PowerShell snap-in. For more information about registering cmdlets, see [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Cmdlet

When your cmdlet has been registered with Windows PowerShell, test it by running it on the command line. For example, test the code for the sample cmdlet. For more information about using cmdlets from the command line, see the [Getting Started with Windows PowerShell](#).

- At the Windows PowerShell prompt, enter the following commands to retrieve the process names through the pipeline.

```
PowerShell  
PS> type ProcessNames | Get-Proc
```

The following output appears.

Handles	NPM(K)	PM(K)	WS(K)	VS(M)	CPU(s)	Id	ProcessName
809	21	40856	4448	147	9.50	2288	iexplore
737	21	26036	16348	144	22.03	3860	iexplore
39	2	1024	388	30	0.08	3396	notepad
3927	62	71836	26984	467	195.19	1848	OUTLOOK

- Enter the following lines to get the process objects that have a `Name` property from the processes called "IEXPLORE". This example uses the `Get-Process` cmdlet (provided by Windows PowerShell) as an upstream command to retrieve the "IEXPLORE" processes.

```
PowerShell  
PS> Get-Process iexplore | Get-Proc
```

The following output appears.

Handles	NPM(K)	PM(K)	WS(K)	VS(M)	CPU(s)	Id	ProcessName
801	21	40720	6544	142	9.52	2288	iexplore
726	21	25872	16652	138	22.09	3860	iexplore
801	21	40720	6544	142	9.52	2288	iexplore
726	21	25872	16652	138	22.09	3860	iexplore

See Also

[Adding Parameters that Process Command Line Input](#)

[Creating Your First Cmdlet](#)

[Extending Object Types and Formatting ↗](#)

[How to Register Cmdlets, Providers, and Host Applications ↗](#)

[Windows PowerShell Reference](#)

[Cmdlet Samples](#)

Adding Non-Terminating Error Reporting to Your Cmdlet

Article • 09/15/2023

Cmdlets can report non-terminating errors by calling the [System.Management.Automation.Cmdlet.WriteError](#) method and still continue to operate on the current input object or on further incoming pipeline objects. This section explains how to create a cmdlet that reports non-terminating errors from its input processing methods.

For non-terminating errors (as well as terminating errors), the cmdlet must pass an [System.Management.Automation.ErrorRecord](#) object identifying the error. Each error record is identified by a unique string called the "error identifier". In addition to the identifier, the category of each error is specified by constants defined by a [System.Management.Automation.ErrorCategory](#) enumeration. The user can view errors based on their category by setting the `$ErrorView` variable to "CategoryView".

For more information about error records, see [Windows PowerShell Error Records](#).

Defining the Cmdlet

The first step in cmdlet creation is always naming the cmdlet and declaring the .NET class that implements the cmdlet. This cmdlet retrieves process information, so the verb name chosen here is "Get". (Almost any sort of cmdlet that is capable of retrieving information can process command-line input.) For more information about approved cmdlet verbs, see [Cmdlet Verb Names](#).

The following is the definition for this `Get-Proc` cmdlet. Details of this definition are given in [Creating Your First Cmdlet](#).

C#

```
[Cmdlet(VerbsCommon.Get, "proc")]
public class GetProcCommand: Cmdlet
```

VB

```
<Cmdlet(VerbsCommon.Get, "Proc")> _
Public Class GetProcCommand
    Inherits Cmdlet
```

Defining Parameters

If necessary, your cmdlet must define parameters for processing input. This Get-Proc cmdlet defines a **Name** parameter as described in [Adding Parameters that Process Command-Line Input](#).

Here is the parameter declaration for the **Name** parameter of this Get-Proc cmdlet.

C#

```
[Parameter(
    Position = 0,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true
)]
[ValidateNotNullOrEmpty]
public string[] Name
{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;
```

VB

```
<Parameter(Position:=0, ValueFromPipeline:=True, _
ValueFromPipelineByPropertyName:=True), ValidateNotNullOrEmpty()> _
Public Property Name() As String()
    Get
        Return processNames
    End Get

    Set(ByVal value As String())
        processNames = value
    End Set

End Property
```

Overriding Input Processing Methods

All cmdlets must override at least one of the input processing methods provided by the `System.Management.Automation.Cmdlet` class. These methods are discussed in [Creating Your First Cmdlet](#).

 Note

Your cmdlet should handle each record as independently as possible.

This Get-Proc cmdlet overrides the [System.Management.Automation.Cmdlet.ProcessRecord](#) method to handle the **Name** parameter for input provided by the user or a script. This method will get the processes for each requested process name or all processes if no name is provided. Details of this override are given in [Creating Your First Cmdlet](#).

Things to Remember When Reporting Errors

The [System.Management.Automation.ErrorRecord](#) object that the cmdlet passes when writing an error requires an exception at its core. Follow the .NET guidelines when determining the exception to use. Basically, if the error is semantically the same as an existing exception, the cmdlet should use or derive from that exception. Otherwise, it should derive a new exception or exception hierarchy directly from the [System.Exception](#) class.

When creating error identifiers (accessed through the `FullyQualifiedErrorId` property of the `ErrorRecord` class) keep the following in mind.

- Use strings that are targeted for diagnostic purposes so that when inspecting the fully qualified identifier you can determine what the error is and where the error came from.
- A well formed fully qualified error identifier might be as follows.

```
CommandNotFoundException,Microsoft.PowerShell.Commands.GetCommandCommand
```

Notice that in the previous example, the error identifier (the first token) designates what the error is and the remaining part indicates where the error came from.

- For more complex scenarios, the error identifier can be a dot separated token that can be parsed on inspection. This allows you too branch on the parts of the error identifier as well as the error identifier and error category.

The cmdlet should assign specific error identifiers to different code paths. Keep the following information in mind for assignment of error identifiers:

- An error identifier should remain constant throughout the cmdlet life cycle. Do not change the semantics of an error identifier between cmdlet versions.
- Use text for an error identifier that tersely corresponds to the error being reported. Do not use white space or punctuation.

- Have your cmdlet generate only error identifiers that are reproducible. For example, it should not generate an identifier that includes a process identifier. Error identifiers are useful to a user only when they correspond to identifiers that are seen by other users experiencing the same problem.

Unhandled exceptions are not caught by PowerShell in the following conditions:

- If a cmdlet creates a new thread and code running in that thread throws an unhandled exception, PowerShell will not catch the error and will terminate the process.
- If an object has code in its destructor or Dispose methods that causes an unhandled exception, PowerShell will not catch the error and will terminate the process.

Reporting Non-terminating Errors

Any one of the input processing methods can report a non-terminating error to the output stream using the [System.Management.Automation.Cmdlet.WriteError](#) method.

Here is a code example from this Get-Proc cmdlet that illustrates the call to [System.Management.Automation.Cmdlet.WriteError](#) from within the override of the [System.Management.Automation.Cmdlet.ProcessRecord](#) method. In this case, the call is made if the cmdlet cannot find a process for a specified process identifier.

C#

```
protected override void ProcessRecord()
{
    // If no name parameter passed to cmdlet, get all processes.
    if (processNames == null)
    {
        WriteObject(Process.GetProcesses(), true);
    }
    else
    {
        // If a name parameter is passed to cmdlet, get and write
        // the associated processes.
        // Write a non-terminating error for failure to retrieve
        // a process.
        foreach (string name in processNames)
        {
            Process[] processes;

            try
            {
                processes = Process.GetProcessesByName(name);
            }

```

```
        catch (InvalidOperationException ex)
    {
        WriteError(new ErrorRecord(
            ex,
            "NameNotFound",
            ErrorCategory.InvalidOperation,
            name));
        continue;
    }

    WriteObject(processes, true);
} // foreach ...
} // else
}
```

Things to Remember About Writing Non-terminating Errors

For a non-terminating error, the cmdlet must generate a specific error identifier for each specific input object.

A cmdlet frequently needs to modify the PowerShell action produced by a non-terminating error. It can do this by defining the `ErrorAction` and `ErrorVariable` parameters. If defining the `ErrorAction` parameter, the cmdlet presents the user options `System.Management.Automation.ActionPreference`, you can also directly influence the action by setting the `$ErrorActionPreference` variable.

The cmdlet can save non-terminating errors to a variable using the `ErrorVariable` parameter, which is not affected by the setting of `ErrorAction`. Failures can be appended to an existing error variable by adding a plus sign (+) to the front of the variable name.

Code Sample

For the complete C# sample code, see [GetProcessSample04 Sample](#).

Define Object Types and Formatting

PowerShell passes information between cmdlets using .NET objects. Consequently, a cmdlet might need to define its own type, or the cmdlet might need to extend an existing type provided by another cmdlet. For more information about defining new types or extending existing types, see [Extending Object Types and Formatting](#).

Building the Cmdlet

After implementing a cmdlet, you must register it with Windows PowerShell through a Windows PowerShell snap-in. For more information about registering cmdlets, see [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Cmdlet

When your cmdlet has been registered with PowerShell, you can test it by running it on the command line. Let's test the sample Get-Proc cmdlet to see whether it reports an error:

- Start PowerShell, and use the Get-Proc cmdlet to retrieve the processes named "TEST".

```
PowerShell  
Get-Proc -Name test
```

The following output appears.

```
Get-Proc : Operation is not valid due to the current state of the  
object.  
At line:1 char:9  
+ Get-Proc <<< -Name test
```

See Also

- [Adding Parameters that Process Pipeline Input](#)
- [Adding Parameters that Process Command-Line Input](#)
- [Creating Your First Cmdlet](#)
- [Extending Object Types and Formatting](#)
- [How to Register Cmdlets, Providers, and Host Applications](#)
- [Windows PowerShell Reference](#)
- [Cmdlet Samples](#)

StopProc Tutorial

Article • 09/15/2023

This section provides a tutorial for creating the Stop-Proc cmdlet, which is very similar to the [Stop-Process](#) cmdlet provided by Windows PowerShell. This tutorial provides fragments of code that illustrate how cmdlets are implemented, and an explanation of the code.

Topics in this Tutorial

The topics in this tutorial are designed to be read sequentially, with each topic building on what was discussed in the previous topic.

- [Creating a Cmdlet that Modifies the System](#): This section describes how to create a cmdlet that supports system modifications, such as stopping a process running on the computer.
- [Adding User Messages to Your Cmdlet](#): This section describes how to add the ability to write user messages, debug messages, warning messages, and progress information to your cmdlet.
- [Adding Aliases, Wildcard Expansion, and Help to Cmdlet Parameters](#): This section describes how to create a cmdlet that supports parameter aliases, Help, and wildcard expansion.
- [Adding Parameter Sets to Cmdlets](#): This section describes how to add parameter sets to a cmdlet. Parameter sets allow the cmdlet to operate differently based on what parameters are specified by the user.

See Also

- [Windows PowerShell SDK](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Creating a Cmdlet that Modifies the System

Article • 09/17/2021

Sometimes a cmdlet must modify the running state of the system, not just the state of the Windows PowerShell runtime. In these cases, the cmdlet should allow the user a chance to confirm whether or not to make the change.

To support confirmation a cmdlet must do two things.

- Declare that the cmdlet supports confirmation when you specify the [System.Management.Automation.CmdletAttribute](#) attribute by setting the `SupportsShouldProcess` keyword to `true`.
- Call [System.Management.Automation.Cmdlet.ShouldProcess](#) during the execution of the cmdlet (as shown in the following example).

By supporting confirmation, a cmdlet exposes the `Confirm` and `WhatIf` parameters that are provided by Windows PowerShell, and also meets the development guidelines for cmdlets (For more information about cmdlet development guidelines, see [Cmdlet Development Guidelines](#)).

Changing the System

The act of "changing the system" refers to any cmdlet that potentially changes the state of the system outside Windows PowerShell. For example, stopping a process, enabling or disabling a user account, or adding a row to a database table are all changes to the system that should be confirmed. In contrast, operations that read data or establish transient connections do not change the system and generally do not require confirmation. Confirmation is also not needed for actions whose effect is limited to inside the Windows PowerShell runtime, such as `Set-Variable`. Cmdlets that might or might not make a persistent change should declare `SupportsShouldProcess` and call [System.Management.Automation.Cmdlet.ShouldProcess](#) only if they are about to make a persistent change.

Note

`ShouldProcess` confirmation applies only to cmdlets. If a command or script modifies the running state of a system by directly calling .NET methods or

properties, or by calling applications outside of Windows PowerShell, this form of confirmation will not be available.

The StopProc Cmdlet

This topic describes a Stop-Proc cmdlet that attempts to stop processes that are retrieved using the Get-Proc cmdlet (described in [Creating Your First Cmdlet](#)).

Defining the Cmdlet

The first step in cmdlet creation is always naming the cmdlet and declaring the .NET class that implements the cmdlet. Because you are writing a cmdlet to change the system, it should be named accordingly. This cmdlet stops system processes, so the verb name chosen here is "Stop", defined by the

[System.Management.Automation.VerbsLifecycle](#) class, with the noun "Proc" to indicate that the cmdlet stops processes. For more information about approved cmdlet verbs, see [Cmdlet Verb Names](#).

The following is the class definition for this Stop-Proc cmdlet.

```
C#  
  
[Cmdlet(VerbsLifecycle.Stop, "Proc",  
        SupportsShouldProcess = true)]  
public class StopProcCommand : Cmdlet
```

Be aware that in the [System.Management.Automation.CmdletAttribute](#) declaration, the `SupportsShouldProcess` attribute keyword is set to `true` to enable the cmdlet to make calls to [System.Management.Automation.Cmdlet.ShouldProcess](#) and [System.Management.Automation.Cmdlet.ShouldContinue](#). Without this keyword set, the `Confirm` and `WhatIf` parameters will not be available to the user.

Extremely Destructive Actions

Some operations are extremely destructive, such as reformatting an active hard disk partition. In these cases, the cmdlet should set `ConfirmImpact` = `ConfirmImpact.High` when declaring the [System.Management.Automation.CmdletAttribute](#) attribute. This setting forces the cmdlet to request user confirmation even when the user has not specified the `Confirm` parameter. However, cmdlet developers should avoid overusing `ConfirmImpact` for operations that are just potentially destructive, such as deleting a

user account. Remember that if `ConfirmImpact` is set to `System.Management.Automation.ConfirmImpact.High`.

Similarly, some operations are unlikely to be destructive, although they do in theory modify the running state of a system outside Windows PowerShell. Such cmdlets can set `ConfirmImpact` to `System.Management.Automation.ConfirmImpact.Low`. This will bypass confirmation requests where the user has asked to confirm only medium-impact and high-impact operations.

Defining Parameters for System Modification

This section describes how to define the cmdlet parameters, including those that are needed to support system modification. See [Adding Parameters that Process CommandLine Input](#) if you need general information about defining parameters.

The Stop-Proc cmdlet defines three parameters: `Name`, `Force`, and `PassThru`.

The `Name` parameter corresponds to the `Name` property of the process input object. Be aware that the `Name` parameter in this sample is mandatory, as the cmdlet will fail if it does not have a named process to stop.

The `Force` parameter allows the user to override calls to `System.Management.Automation.Cmdlet.ShouldContinue`. In fact, any cmdlet that calls `System.Management.Automation.Cmdlet.ShouldContinue` should have a `Force` parameter so that when `Force` is specified, the cmdlet skips the call to `System.Management.Automation.Cmdlet.ShouldContinue` and proceeds with the operation. Be aware that this does not affect calls to `System.Management.Automation.Cmdlet.ShouldProcess`.

The `PassThru` parameter allows the user to indicate whether the cmdlet passes an output object through the pipeline, in this case, after a process is stopped. Be aware that this parameter is tied to the cmdlet itself instead of to a property of the input object.

Here is the parameter declaration for the Stop-Proc cmdlet.

```
C#  
  
[Parameter()  
    Position = 0,  
    Mandatory = true,  
    ValueFromPipeline = true,  
    ValueFromPipelineByPropertyName = true  
)]  
public string[] Name
```

```

{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;

/// <summary>
/// Specify the Force parameter that allows the user to override
/// the ShouldContinue call to force the stop operation. This
/// parameter should always be used with caution.
/// </summary>
[Parameter]
public SwitchParameter Force
{
    get { return force; }
    set { force = value; }
}
private bool force;

/// <summary>
/// Specify the PassThru parameter that allows the user to specify
/// that the cmdlet should pass the process object down the pipeline
/// after the process has been stopped.
/// </summary>
[Parameter]
public SwitchParameter PassThru
{
    get { return passThru; }
    set { passThru = value; }
}
private bool passThru;

```

Overriding an Input Processing Method

The cmdlet must override an input processing method. The following code illustrates the `System.Management.Automation.Cmdlet.ProcessRecord` override used in the sample Stop-Proc cmdlet. For each requested process name, this method ensures that the process is not a special process, tries to stop the process, and then sends an output object if the `PassThru` parameter is specified.

C#

```

protected override void ProcessRecord()
{
    foreach (string name in processNames)
    {
        // For every process name passed to the cmdlet, get the associated
        // process(es). For failures, write a non-terminating error
        Process[] processes;

```

```

try
{
    processes = Process.GetProcessesByName(name);
}
catch (InvalidOperationException ioe)
{
    WriteError(new ErrorRecord(ioe, "Unable to access the target process by
name",
                           ErrorCategory.InvalidOperation, name));
    continue;
}

// Try to stop the process(es) that have been retrieved for a name
foreach (Process process in processes)
{
    string processName;

    try
    {
        processName = process.ProcessName;
    }

    catch (Win32Exception e)
    {
        WriteError(new ErrorRecord(e, "ProcessNameNotFound",
                                   ErrorCategory.ReadError, process));
        continue;
    }

    // Call Should Process to confirm the operation first.
    // This is always false if WhatIf is set.
    if (!ShouldProcess(string.Format("{0} ({1})", processName,
                                    process.Id)))
    {
        continue;
    }
    // Call ShouldContinue to make sure the user really does want
    // to stop a critical process that could possibly stop the computer.
    bool criticalProcess =
        criticalProcessNames.Contains(processName.ToLower());

    if (criticalProcess &&!force)
    {
        string message = String.Format
            ("The process \"{0}\" is a critical process and should not
be stopped. Are you sure you wish to stop the process?",
            processName);

        // It is possible that ProcessRecord is called multiple times
        // when the Name parameter receives objects as input from the
        // pipeline. So to retain YesToAll and NoToAll input that the
        // user may enter across multiple calls to ProcessRecord, this
        // information is stored as private members of the cmdlet.
        if (!ShouldContinue(message, "Warning!",
                           ref yesToAll,

```

```

                    ref noToAll))
{
    continue;
}
} // if (criticalProcess...
// Stop the named process.
try
{
    process.Kill();
}
catch (Exception e)
{
    if ((e is Win32Exception) || (e is SystemException) ||
        (e is InvalidOperationException))
    {
        // This process could not be stopped so write
        // a non-terminating error.
        string message = String.Format("{0} {1} {2}",
            "Could not stop process \\"", processName,
            "\\.");
        WriteError(new ErrorRecord(e, message,
            ErrorCategory.CloseError, process));
        continue;
    } // if ((e is...
    else throw;
} // catch

// If the PassThru parameter argument is
// True, pass the terminated process on.
if (passThru)
{
    WriteObject(process);
}
} // foreach (Process...
} // foreach (string...
} // ProcessRecord

```

Calling the ShouldProcess Method

The input processing method of your cmdlet should call the [System.Management.Automation.Cmdlet.ShouldProcess](#) method to confirm execution of an operation before a change (for example, deleting files) is made to the running state of the system. This allows the Windows PowerShell runtime to supply the correct "WhatIf" and "Confirm" behavior within the shell.

ⓘ Note

If a cmdlet states that it supports should process and fails to make the [System.Management.Automation.Cmdlet.ShouldProcess](#) call, the user might

modify the system unexpectedly.

The call to [System.Management.Automation.Cmdlet.ShouldProcess](#) sends the name of the resource to be changed to the user, with the Windows PowerShell runtime taking into account any command-line settings or preference variables in determining what should be displayed to the user.

The following example shows the call to

[System.Management.Automation.Cmdlet.ShouldProcess](#) from the override of the [System.Management.Automation.Cmdlet.ProcessRecord](#) method in the sample Stop-Proc cmdlet.

```
C#
```

```
if (!ShouldProcess(string.Format("{0} ({1})", processName,
                                 process.Id)))
{
    continue;
}
```

Calling the ShouldContinue Method

The call to the [System.Management.Automation.Cmdlet.ShouldContinue](#) method sends a secondary message to the user. This call is made after the call to [System.Management.Automation.Cmdlet.ShouldProcess](#) returns `true` and if the `Force` parameter was not set to `true`. The user can then provide feedback to say whether the operation should be continued. Your cmdlet calls [System.Management.Automation.Cmdlet.ShouldContinue](#) as an additional check for potentially dangerous system modifications or when you want to provide yes-to-all and no-to-all options to the user.

The following example shows the call to

[System.Management.Automation.Cmdlet.ShouldContinue](#) from the override of the [System.Management.Automation.Cmdlet.ProcessRecord](#) method in the sample Stop-Proc cmdlet.

```
C#
```

```
if (criticalProcess &&!force)
{
    string message = String.Format
        ("The process \"{0}\" is a critical process and should not be
         stopped. Are you sure you wish to stop the process?",
         processName);
```

```
// It is possible that ProcessRecord is called multiple times
// when the Name parameter receives objects as input from the
// pipeline. So to retain YesToAll and NoToAll input that the
// user may enter across multiple calls to ProcessRecord, this
// information is stored as private members of the cmdlet.
if (!ShouldContinue(message, "Warning!",
                     ref yesToAll,
                     ref noToAll))
{
    continue;
}
} // if (criticalProcess...
```

Stopping Input Processing

The input processing method of a cmdlet that makes system modifications must provide a way of stopping the processing of input. In the case of this Stop-Proc cmdlet, a call is made from the [System.Management.Automation.Cmdlet.ProcessRecord](#) method to the [System.Diagnostics.Process.Kill](#)* method. Because the `PassThru` parameter is set to `true`, [System.Management.Automation.Cmdlet.ProcessRecord](#) also calls [System.Management.Automation.Cmdlet.WriteObject](#) to send the process object to the pipeline.

Code Sample

For the complete C# sample code, see [StopProcessSample01 Sample](#).

Defining Object Types and Formatting

Windows PowerShell passes information between cmdlets using .NET objects. Consequently, a cmdlet may need to define its own type, or the cmdlet may need to extend an existing type provided by another cmdlet. For more information about defining new types or extending existing types, see [Extending Object Types and Formatting](#) ↗.

Building the Cmdlet

After implementing a cmdlet, it must be registered with Windows PowerShell through a Windows PowerShell snap-in. For more information about registering cmdlets, see [How to Register Cmdlets, Providers, and Host Applications](#) ↗.

Testing the Cmdlet

When your cmdlet has been registered with Windows PowerShell, you can test it by running it on the command line. Here are several tests that test the Stop-Proc cmdlet. For more information about using cmdlets from the command line, see the [Getting Started with Windows PowerShell](#).

- Start Windows PowerShell and use the Stop-Proc cmdlet to stop processing as shown below. Because the cmdlet specifies the `Name` parameter as mandatory, the cmdlet queries for the parameter.

```
PowerShell
```

```
PS> Stop-Proc
```

The following output appears.

```
Cmdlet Stop-Proc at command pipeline position 1  
Supply values for the following parameters:  
Name[0]:
```

- Now let's use the cmdlet to stop the process named "NOTEPAD". The cmdlet asks you to confirm the action.

```
PowerShell
```

```
PS> Stop-Proc -Name notepad
```

The following output appears.

```
Confirm  
Are you sure you want to perform this action?  
Performing operation "Stop-Proc" on Target "notepad (4996)".  
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"): Y
```

- Use Stop-Proc as shown to stop the critical process named "WINLOGON". You are prompted and warned about performing this action because it will cause the operating system to reboot.

PowerShell

```
PS> Stop-Proc -Name Winlogon
```

The following output appears.

Output

```
Confirm  
Are you sure you want to perform this action?  
Performing operation "Stop-Proc" on Target "winlogon (656)".  
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"): Y  
Warning!  
The process " winlogon " is a critical process and should not be  
stopped. Are you sure you wish to stop the process?  
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"): N
```

- Let's now try to stop the WINLOGON process without receiving a warning. Be aware that this command entry uses the `Force` parameter to override the warning.

PowerShell

```
PS> Stop-Proc -Name winlogon -Force
```

The following output appears.

Output

```
Confirm  
Are you sure you want to perform this action?  
Performing operation "Stop-Proc" on Target "winlogon (656)".  
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"): N
```

See Also

[Adding Parameters that Process Command-Line Input](#)

[Extending Object Types and Formatting ↗](#)

[How to Register Cmdlets, Providers, and Host Applications ↗](#)

[Windows PowerShell SDK](#)

Cmdlet Samples

Adding User Messages to Your Cmdlet

Article • 09/17/2021

Cmdlets can write several kinds of messages that can be displayed to the user by the Windows PowerShell runtime. These messages include the following types:

- Verbose messages that contain general user information.
- Debug messages that contain troubleshooting information.
- Warning messages that contain a notification that the cmdlet is about to perform an operation that can have unexpected results.
- Progress report messages that contain information about how much work the cmdlet has completed when performing an operation that takes a long time.

There are no limits to the number of messages that your cmdlet can write or the type of messages that your cmdlet writes. Each message is written by making a specific call from within the input processing method of your cmdlet.

Defining the Cmdlet

The first step in cmdlet creation is always naming the cmdlet and declaring the .NET class that implements the cmdlet. Any sort of cmdlet can write user notifications from its input processing methods; so, in general, you can name this cmdlet using any verb that indicates what system modifications the cmdlet performs. For more information about approved cmdlet verbs, see [Cmdlet Verb Names](#).

The Stop-Proc cmdlet is designed to modify the system; therefore, the [System.Management.Automation.CmdletAttribute](#) declaration for the .NET class must include the `SupportsShouldProcess` attribute keyword and be set to `true`.

The following code is the definition for this Stop-Proc cmdlet class. For more information about this definition, see [Creating a Cmdlet that Modifies the System](#).

C#

```
[Cmdlet(VerbsLifecycle.Stop, "proc",
        SupportsShouldProcess = true)]
public class StopProcCommand : Cmdlet
```

Defining Parameters for System Modification

The Stop-Proc cmdlet defines three parameters: `Name`, `Force`, and `PassThru`. For more information about defining these parameters, see [Creating a Cmdlet that Modifies the System](#).

Here is the parameter declaration for the Stop-Proc cmdlet.

C#

```
[Parameter(
    Position = 0,
    Mandatory = true,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true
)]
public string[] Name
{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;

/// <summary>
/// Specify the Force parameter that allows the user to override
/// the ShouldContinue call to force the stop operation. This
/// parameter should always be used with caution.
/// </summary>
[Parameter]
public SwitchParameter Force
{
    get { return force; }
    set { force = value; }
}
private bool force;

/// <summary>
/// Specify the PassThru parameter that allows the user to specify
/// that the cmdlet should pass the process object down the pipeline
/// after the process has been stopped.
/// </summary>
[Parameter]
public SwitchParameter PassThru
{
    get { return passThru; }
    set { passThru = value; }
}
private bool passThru;
```

Overriding an Input Processing Method

Your cmdlet must override an input processing method, most often it will be [System.Management.Automation.Cmdlet.ProcessRecord](#). This Stop-Proc cmdlet overrides the [System.Management.Automation.Cmdlet.ProcessRecord](#) input processing method. In this implementation of the Stop-Proc cmdlet, calls are made to write verbose messages, debug messages, and warning messages.

 **Note**

For more information about how this method calls the [System.Management.Automation.Cmdlet.ShouldProcess](#) and [System.Management.Automation.Cmdlet.ShouldContinue](#) methods, see [Creating a Cmdlet that Modifies the System](#).

Writing a Verbose Message

The [System.Management.Automation.Cmdlet.WriteVerbose](#) method is used to write general user-level information that is unrelated to specific error conditions. The system administrator can then use that information to continue processing other commands. In addition, any information written using this method should be localized as needed.

The following code from this Stop-Proc cmdlet shows two calls to the [System.Management.Automation.Cmdlet.WriteVerbose](#) method from the override of the [System.Management.Automation.Cmdlet.ProcessRecord](#) method.

C#

```
message = String.Format("Attempting to stop process \'{0}\'.", name);
WriteVerbose(message);
```

C#

```
message = String.Format("Stopped process \'{0}\', pid {1}.",
processName, process.Id);

WriteVerbose(message);
```

Writing a Debug Message

The [System.Management.Automation.Cmdlet.WriteDebug](#) method is used to write debug messages that can be used to troubleshoot the operation of the cmdlet. The call is made from an input processing method.

Note

Windows PowerShell also defines a `Debug` parameter that presents both verbose and debug information. If your cmdlet supports this parameter, it does not need to call `System.Management.Automation.Cmdlet.WriteDebug` in the same code that calls `System.Management.Automation.Cmdlet.WriteVerbose`.

The following two sections of code from the sample Stop-Proc cmdlet show calls to the `System.Management.Automation.Cmdlet.WriteDebug` method from the override of the `System.Management.Automation.Cmdlet.ProcessRecord` method.

This debug message is written immediately before `System.Management.Automation.Cmdlet.ShouldProcess` is called.

```
C#
```

```
message =
    String.Format("Acquired name for pid {0} : \'{1}\'",
        process.Id, processName);
WriteDebug(message);
```

This debug message is written immediately before `System.Management.Automation.Cmdlet.WriteObject` is called.

```
C#
```

```
message =
    String.Format("Writing process \'{0}\' to pipeline",
        processName);
WriteDebug(message);
WriteObject(process);
```

Windows PowerShell automatically routes any `System.Management.Automation.Cmdlet.WriteDebug` calls to the tracing infrastructure and cmdlets. This allows the method calls to be traced to the hosting application, a file, or a debugger without your having to do any extra development work within the cmdlet. The following command-line entry implements a tracing operation.

```
PS> Trace-Expression Stop-Proc -File proc.log -Command Stop-Proc notepad
```

Writing a Warning Message

The [System.Management.Automation.Cmdlet.WriteWarning](#) method is used to write a warning when the cmdlet is about to perform an operation that might have an unexpected result, for example, overwriting a read-only file.

The following code from the sample Stop-Proc cmdlet shows the call to the [System.Management.Automation.Cmdlet.WriteWarning](#) method from the override of the [System.Management.Automation.Cmdlet.ProcessRecord](#) method.

```
C#
```

```
if (criticalProcess)
{
    message =
        String.Format("Stopping the critical process \">{0}\",
                      processName);
    WriteWarning(message);
} // if (criticalProcess...
```

Writing a Progress Message

The [System.Management.Automation.Cmdlet.WriteProgress](#) is used to write progress messages when cmdlet operations take an extended amount of time to complete. A call to [System.Management.Automation.Cmdlet.WriteProgress](#) passes a [System.Management.Automation.ProgressRecord](#) object that is sent to the hosting application for rendering to the user.

 **Note**

This Stop-Proc cmdlet does not include a call to the [System.Management.Automation.Cmdlet.WriteProgress](#) method.

The following code is an example of a progress message written by a cmdlet that is attempting to copy an item.

```
C#
```

```
int myId = 0;
string myActivity = "Copy-item: Copying *.* to C:\abc";
string myStatus = "Copying file bar.txt";
ProgressRecord pr = new ProgressRecord(myId, myActivity, myStatus);
WriteProgress(pr);

pr.RecordType = ProgressRecordType.Completed;
WriteProgress(pr);
```

Code Sample

For the complete C# sample code, see [StopProcessSample02 Sample](#).

Define Object Types and Formatting

Windows PowerShell passes information between cmdlets using .NET objects. Consequently, a cmdlet might need to define its own type, or the cmdlet might need to extend an existing type provided by another cmdlet. For more information about defining new types or extending existing types, see [Extending Object Types and Formatting](#).

Building the Cmdlet

After implementing a cmdlet, it must be registered with Windows PowerShell through a Windows PowerShell snap-in. For more information about registering cmdlets, see [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Cmdlet

When your cmdlet has been registered with Windows PowerShell, you can test it by running it on the command line. Let's test the sample Stop-Proc cmdlet. For more information about using cmdlets from the command line, see the [Getting Started with Windows PowerShell](#).

- The following command-line entry uses Stop-Proc to stop the process named "NOTEPAD", provide verbose notifications, and print debug information.

```
PowerShell
```

```
PS> Stop-Proc -Name notepad -Verbose -Debug
```

The following output appears.

```
VERBOSE: Attempting to stop process " notepad ".
DEBUG: Acquired name for pid 5584 : "notepad"

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help
```

```
(default is "Y"): Y

Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Proc" on Target "notepad (5584)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"): Y
VERBOSE: Stopped process "notepad", pid 5584.
```

See Also

[Create a Cmdlet that Modifies the System](#)

[How to Create a Windows PowerShell Cmdlet](#)

[Extending Object Types and Formatting ↗](#)

[How to Register Cmdlets, Providers, and Host Applications ↗](#)

[Windows PowerShell SDK](#)

Adding Aliases, Wildcard Expansion, and Help to Cmdlet Parameters

Article • 05/12/2022

This section describes how to add aliases, wildcard expansion, and Help messages to the parameters of the `Stop-Proc` cmdlet (described in [Creating a Cmdlet that Modifies the System](#)).

This `Stop-Proc` cmdlet attempts to stop processes that are retrieved using the `Get-Proc` cmdlet (described in [Creating Your First Cmdlet](#)).

Defining the Cmdlet

The first step in cmdlet creation is always naming the cmdlet and declaring the .NET class that implements the cmdlet. Because you are writing a cmdlet to change the system, it should be named accordingly. Because this cmdlet stops system processes, it uses the verb **Stop**, defined by the `System.Management.Automation.VerbsLifecycle` class, with the noun **Proc** to indicate process. For more information about approved cmdlet verbs, see [Cmdlet Verb Names](#).

The following code is the class definition for this `Stop-Proc` cmdlet.

```
C#  
  
[Cmdlet(VerbsLifecycle.Stop, "proc",  
        SupportsShouldProcess = true)]  
public class StopProcCommand : Cmdlet
```

Defining Parameters for System Modification

Your cmdlet needs to define parameters that support system modifications and user feedback. The cmdlet should define a **Name** parameter or equivalent so that the cmdlet will be able to modify the system by some sort of identifier. In addition, the cmdlet should define the **Force** and **PassThru** parameters. For more information about these parameters, see [Creating a Cmdlet that Modifies the System](#).

Defining a Parameter Alias

A parameter alias can be an alternate name or a well-defined 1-letter or 2-letter short name for a cmdlet parameter. In both cases, the goal of using aliases is to simplify user entry from the command line. Windows PowerShell supports parameter aliases through the [System.Management.Automation.AliasAttribute](#) attribute, which uses the declaration syntax `[Alias()]`.

The following code shows how an alias is added to the **Name** parameter.

```
C#  
  
/// <summary>  
/// Specify the mandatory Name parameter used to identify the  
/// processes to be stopped.  
/// </summary>  
[Parameter(  
    Position = 0,  
    Mandatory = true,  
    ValueFromPipeline = true,  
    ValueFromPipelineByPropertyName = true,  
    HelpMessage = "The name of one or more processes to stop.  
Wildcards are permitted."  
)  
[Alias("ProcessName")]  
public string[] Name  
{  
    get { return processNames; }  
    set { processNames = value; }  
}  
private string[] processNames;
```

In addition to using the [System.Management.Automation.AliasAttribute](#) attribute, the Windows PowerShell runtime performs partial name matching, even if no aliases are specified. For example, if your cmdlet has a **FileName** parameter and that is the only parameter that starts with `F`, the user could enter `Filename`, `Filenam`, `File`, `Fi`, or `F` and still recognize the entry as the **FileName** parameter.

Creating Help for Parameters

Windows PowerShell allows you to create Help for cmdlet parameters. Do this for any parameter used for system modification and user feedback. For each parameter to support Help, you can set the **HelpMessage** attribute keyword in the [System.Management.Automation.ParameterAttribute](#) attribute declaration. This keyword defines the text to display to the user for assistance in using the parameter. You can also set the **HelpMessageBaseName** keyword to identify the base name of a resource to use

for the message. If you set this keyword, you must also set the **HelpMessageResourceId** keyword to specify the resource identifier.

The following code from this `Stop-Proc` cmdlet defines the **HelpMessage** attribute keyword for the **Name** parameter.

```
C#
```

```
/// <summary>
/// Specify the mandatory Name parameter used to identify the
/// processes to be stopped.
/// </summary>
[Parameter(
    Position = 0,
    Mandatory = true,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true,
    HelpMessage = "The name of one or more processes to stop.
Wildcards are permitted."
)]
```

Overriding an Input Processing Method

Your cmdlet must override an input processing method, most often this will be `System.Management.Automation.Cmdlet.ProcessRecord`. When modifying the system, the cmdlet should call the `System.Management.Automation.Cmdlet.ShouldProcess` and `System.Management.Automation.Cmdlet.ShouldContinue` methods to allow the user to provide feedback before a change is made. For more information about these methods, see [Creating a Cmdlet that Modifies the System](#).

Supporting Wildcard Expansion

To allow the selection of multiple objects, your cmdlet can use the `System.Management.Automation.WildcardPattern` and `System.Management.Automation.WildcardOptions` classes to provide wildcard expansion support for parameter input. Examples of wildcard patterns are `1sa*`, `*.txt`, and `[a-c]*`. Use the back-quote character (`\``) as an escape character when the pattern contains a character that should be used literally.

Wildcard expansions of file and path names are examples of common scenarios where the cmdlet may want to allow support for path inputs when the selection of multiple objects is required. A common case is in the file system, where a user wants to see all files residing in the current folder.

You should need a customized wildcard pattern matching implementation only rarely. In this case, your cmdlet should support either the full POSIX 1003.2, 3.13 specification for wildcard expansion or the following simplified subset:

- **Question mark (?)**. Matches any character at the specified location.
- **Asterisk (*)**. Matches zero or more characters starting at the specified location.
- **Open bracket ([)**. Introduces a pattern bracket expression that can contain characters or a range of characters. If a range is required, a hyphen (-) is used to indicate the range.
- **Close bracket (])**. Ends a pattern bracket expression.
- **Back-quote escape character (`)**. Indicates that the next character should be taken literally. Be aware that when specifying the back-quote character from the command line (as opposed to specifying it programmatically), the back-quote escape character must be specified twice.

Note

For more information about wildcard patterns, see [Supporting Wildcards in Cmdlet Parameters](#).

The following code shows how to set wildcard options and define the wildcard pattern used for resolving the **Name** parameter for this cmdlet.

```
C#
```

```
WildcardOptions options = WildcardOptions.IgnoreCase |  
    WildcardOptions.Compiled;  
WildcardPattern wildcard = new WildcardPattern(name,options);
```

The following code shows how to test whether the process name matches the defined wildcard pattern. Notice that, in this case, if the process name does not match the pattern, the cmdlet continues on to get the next process name.

```
C#
```

```
if (!wildcard.IsMatch(processName))  
{  
    continue;  
}
```

Code Sample

For the complete C# sample code, see [StopProcessSample03 Sample](#).

Define Object Types and Formatting

Windows PowerShell passes information between cmdlets using .NET objects. Consequently, a cmdlet may need to define its own type, or the cmdlet may need to extend an existing type provided by another cmdlet. For more information about defining new types or extending existing types, see [Extending Object Types and Formatting](#).

Building the Cmdlet

After implementing a cmdlet, it must be registered with Windows PowerShell through a Windows PowerShell snap-in. For more information about registering cmdlets, see [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Cmdlet

When your cmdlet has been registered with Windows PowerShell, you can test it by running it on the command line. Let's test the sample Stop-Proc cmdlet. For more information about using cmdlets from the command line, see the [Getting Started with Windows PowerShell](#).

- Start Windows PowerShell and use `Stop-Proc` to stop a process using the `ProcessName` alias for the `Name` parameter.

```
PowerShell  
PS> Stop-Proc -ProcessName notepad
```

The following output appears.

```
Confirm  
Are you sure you want to perform this action?  
Performing operation "Stop-Proc" on Target "notepad (3496)".  
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help  
(default is "Y"): Y
```

- Make the following entry on the command line. Because the `Name` parameter is mandatory, you are prompted for it. Entering `!?` brings up the help text associated

with the parameter.

```
PowerShell
```

```
PS> Stop-Proc
```

The following output appears.

```
Cmdlet Stop-Proc at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
Name[0]: !?
The name of one or more processes to stop. Wildcards are permitted.
Name[0]: notepad
```

- Now make the following entry to stop all processes that match the wildcard pattern `*note*`. You are prompted before stopping each process that matches the pattern.

```
PowerShell
```

```
PS> Stop-Proc -Name *note*
```

The following output appears.

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Proc" on Target "notepad (1112)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"): Y
```

The following output appears.

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Proc" on Target "ONENOTEM (3712)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"): N
```

The following output appears.

Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Proc" on Target "ONENOTE (3592)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"): N

See Also

- [Create a Cmdlet that Modifies the System](#)
- [How to Create a Windows PowerShell Cmdlet](#)
- [Extending Object Types and Formatting ↗](#)
- [How to Register Cmdlets, Providers, and Host Applications ↗](#)
- [Supporting Wildcards in Cmdlet Parameters](#)
- [Windows PowerShell SDK](#)

Adding Parameter Sets to a Cmdlet

Article • 09/17/2021

Things to Know About Parameter Sets

Windows PowerShell defines a parameter set as a group of parameters that operate together. By grouping the parameters of a cmdlet, you can create a single cmdlet that can change its functionality based on what group of parameters the user specifies.

An example of a cmdlet that uses two parameter sets to define different functionalities is the `Get-EventLog` cmdlet that is provided by Windows PowerShell. This cmdlet returns different information when the user specifies the `List` or `LogName` parameter. If the `LogName` parameter is specified, the cmdlet returns information about the events in a given event log. If the `List` parameter is specified, the cmdlet returns information about the log files themselves (not the event information they contain). In this case, the `List` and `LogName` parameters identify two separate parameter sets.

Two important things to remember about parameter sets is that the Windows PowerShell runtime uses only one parameter set for a particular input, and that each parameter set must have at least one parameter that is unique for that parameter set.

To illustrate that last point, this `Stop-Proc` cmdlet uses three parameter sets:

`ProcessName`, `ProcessId`, and `InputObject`. Each of these parameter sets has one parameter that is not in the other parameter sets. The parameter sets could share other parameters, but the cmdlet uses the unique parameters `ProcessName`, `ProcessId`, and `InputObject` to identify which set of parameters that the Windows PowerShell runtime should use.

Declaring the Cmdlet Class

The first step in cmdlet creation is always naming the cmdlet and declaring the .NET class that implements the cmdlet. For this cmdlet, the lifecycle verb "Stop" is used because the cmdlet stops system processes. The noun name "Proc" is used because the cmdlet works on processes. In the declaration below, note that the cmdlet verb and noun name are reflected in the name of the cmdlet class.

Note

For more information about approved cmdlet verb names, see [Cmdlet Verb Names](#).

The following code is the class definition for this Stop-Proc cmdlet.

C#

```
[Cmdlet(VerbsLifecycle.Stop, "Proc",
        DefaultParameterSetName = "ProcessId",
        SupportsShouldProcess = true)]
public class StopProcCommand : PSCmdlet
```

VB

```
<Cmdlet(VerbsLifecycle.Stop, "Proc", DefaultParameterSetName:="ProcessId", _
SupportsShouldProcess:=True)> _
Public Class StopProcCommand
    Inherits PSCmdlet
```

Declaring the Parameters of the Cmdlet

This cmdlet defines three parameters needed as input to the cmdlet (these parameters also define the parameter sets), as well as a `Force` parameter that manages what the cmdlet does and a `PassThru` parameter that determines whether the cmdlet sends an output object through the pipeline. By default, this cmdlet does not pass an object through the pipeline. For more information about these last two parameters, see [Creating a Cmdlet that Modifies the System](#).

Declaring the Name Parameter

This input parameter allows the user to specify the names of the processes to be stopped. Note that the `ParameterSetName` attribute keyword of the `System.Management.Automation.ParameterAttribute` attribute specifies the `ProcessName` parameter set for this parameter.

C#

```
[Parameter(
    Position = 0,
    ParameterSetName = "ProcessName",
    Mandatory = true,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true,
```

```

        HelpMessage = "The name of one or more processes to stop. Wildcards are
permitted."
    )]
[Alias("ProcessName")]
public string[] Name
{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;

```

VB

```

<Parameter(Position:=0, ParameterSetName:="ProcessName", _
Mandatory:=True, _
ValueFromPipeline:=True, ValueFromPipelineByPropertyName:=True, _
HelpMessage:="The name of one or more processes to stop. " & _
    "Wildcards are permitted."), [Alias]("ProcessName")> _
Public Property Name() As String()
    Get
        Return processNames
    End Get
    Set(ByVal value As String())
        processNames = value
    End Set
End Property

Private processNames() As String

```

Note also that the alias "ProcessName" is given to this parameter.

Declaring the Id Parameter

This input parameter allows the user to specify the identifiers of the processes to be stopped. Note that the `ParameterSetName` attribute keyword of the `System.Management.Automation.ParameterAttribute` attribute specifies the `ProcessId` parameter set.

C#

```

[Parameter(
    ParameterSetName = "ProcessId",
    Mandatory = true,
    ValueFromPipelineByPropertyName = true,
    ValueFromPipeline = true
)]
[Alias("ProcessId")]
public int[] Id
{

```

```

    get { return processIds; }
    set { processIds = value; }
}
private int[] processIds;

```

VB

```

<Parameter(ParameterSetName:="ProcessId", _
Mandatory:=True, _
ValueFromPipelineByPropertyName:=True, _
ValueFromPipeline:=True), [Alias]("ProcessId")> _
Public Property Id() As Integer()
    Get
        Return processIds
    End Get
    Set(ByVal value As Integer())
        processIds = value
    End Set
End Property
Private processIds() As Integer

```

Note also that the alias "ProcessId" is given to this parameter.

Declaring the InputObject Parameter

This input parameter allows the user to specify an input object that contains information about the processes to be stopped. Note that the `ParameterSetName` attribute keyword of the `System.Management.Automation.ParameterAttribute` attribute specifies the `InputObject` parameter set for this parameter.

C#

```

[Parameter(
    ParameterSetName = "InputObject",
    Mandatory = true,
    ValueFromPipeline = true)]
public Process[] InputObject
{
    get { return inputObject; }
    set { inputObject = value; }
}
private Process[] inputObject;

```

VB

```

<Parameter(ParameterSetName:="InputObject", _
Mandatory:=True, ValueFromPipeline:=True)> _
Public Property InputObject() As Process()

```

```
Get
    Return myInputObject
End Get
Set(ByVal value As Process())
    myInputObject = value
End Set
End Property
Private myInputObject() As Process
```

Note also that this parameter has no alias.

Declaring Parameters in Multiple Parameter Sets

Although there must be a unique parameter for each parameter set, parameters can belong to more than one parameter set. In these cases, give the shared parameter a [System.Management.Automation.ParameterAttribute](#) attribute declaration for each set to which that the parameter belongs. If a parameter is in all parameter sets, you only have to declare the parameter attribute once and do not need to specify the parameter set name.

Overriding an Input Processing Method

Every cmdlet must override an input processing method, most often this will be the [System.Management.Automation.Cmdlet.ProcessRecord](#) method. In this cmdlet, the [System.Management.Automation.Cmdlet.ProcessRecord](#) method is overridden so that the cmdlet can process any number of processes. It contains a Select statement that calls a different method based on which parameter set the user has specified.

```
C#
protected override void ProcessRecord()
{
    switch (ParameterSetName)
    {
        case "ProcessName":
            ProcessByName();
            break;

        case "ProcessId":
            ProcessById();
            break;

        case "InputObject":
            foreach (Process process in inputObject)
            {
                SafeStopProcess(process);
            }
    }
}
```

```
        break;

    default:
        throw new ArgumentException("Bad ParameterSet Name");
    } // switch (ParameterSetName...
} // ProcessRecord
```

VB

```
Protected Overrides Sub ProcessRecord()
    Select Case ParameterSetName
        Case "ProcessName"
            ProcessByName()

        Case "ProcessId"
            ProcessById()

        Case "InputObject"
            Dim process As Process
            For Each process In myInputObject
                SafeStopProcess(process)
            Next process

        Case Else
            Throw New ArgumentException("Bad ParameterSet Name")
    End Select

End Sub 'ProcessRecord ' ProcessRecord
```

The Helper methods called by the Select statement are not described here, but you can see their implementation in the complete code sample in the next section.

Code Sample

For the complete C# sample code, see [StopProcessSample04 Sample](#).

Defining Object Types and Formatting

Windows PowerShell passes information between cmdlets using .NET objects. Consequently, a cmdlet might need to define its own type, or the cmdlet might need to extend an existing type provided by another cmdlet. For more information about defining new types or extending existing types, see [Extending Object Types and Formatting](#).

Building the Cmdlet

After implementing a cmdlet, you must register it with Windows PowerShell through a Windows PowerShell snap-in. For more information about registering cmdlets, see [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Cmdlet

When your cmdlet has been registered with Windows PowerShell, test it by running it on the command line. Here are some tests that show how the `ProcessId` and `InputObject` parameters can be used to test their parameter sets to stop a process.

- With Windows PowerShell started, run the `Stop-Proc` cmdlet with the `ProcessId` parameter set to stop a process based on its identifier. In this case, the cmdlet is using the `ProcessId` parameter set to stop the process.

```
PS> Stop-Proc -Id 444
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Proc" on Target "notepad (444)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"): Y
```

- With Windows PowerShell started, run the `Stop-Proc` cmdlet with the `InputObject` parameter set to stop processes on the Notepad object retrieved by the `Get-Process` command.

```
PS> Get-Process notepad | Stop-Proc
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Proc" on Target "notepad (444)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"): N
```

See Also

[Creating a Cmdlet that Modifies the System](#)

[How to Create a Windows PowerShell Cmdlet](#)

[Extending Object Types and Formatting](#)

[How to Register Cmdlets, Providers, and Host Applications](#)

[Windows PowerShell SDK](#)

SelectStr Tutorial

Article • 09/17/2021

This section provides a tutorial for creating the Select-Str cmdlet, which is very similar to the [Select-String](#) cmdlet provided by Windows PowerShell. This tutorial provides fragments of code that illustrate how cmdlets are implemented, and an explanation of the code.

Topic in this Tutorial

[Creating a Cmdlet to Access a Data Store](#) This section describes how to create a cmdlet that selects strings that are in a file or object.

See Also

[Creating a Cmdlet to Access a Data Store](#)

[Windows PowerShell SDK](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Creating a Cmdlet to Access a Data Store

Article • 09/17/2021

This section describes how to create a cmdlet that accesses stored data by way of a Windows PowerShell provider. This type of cmdlet uses the Windows PowerShell provider infrastructure of the Windows PowerShell runtime and, therefore, the cmdlet class must derive from the [System.Management.Automation.PSCmdlet](#) base class.

The Select-Str cmdlet described here can locate and select strings in a file or object. The patterns used to identify the string can be specified explicitly through the `Path` parameter of the cmdlet or implicitly through the `Script` parameter.

The cmdlet is designed to use any Windows PowerShell provider that derives from [System.Management.Automation.Provider.IContentCmdletProvider](#). For example, the cmdlet can specify the FileSystem provider or the Variable provider that is provided by Windows PowerShell. For more information about Windows PowerShell providers, see [Designing Your Windows PowerShell provider](#).

Defining the Cmdlet Class

The first step in cmdlet creation is always naming the cmdlet and declaring the .NET class that implements the cmdlet. This cmdlet detects certain strings, so the verb name chosen here is "Select", defined by the [System.Management.Automation.VerbsCommon](#) class. The noun name "Str" is used because the cmdlet acts upon strings. In the declaration below, note that the cmdlet verb and noun name are reflected in the name of the cmdlet class. For more information about approved cmdlet verbs, see [Cmdlet Verb Names](#).

The .NET class for this cmdlet must derive from the [System.Management.Automation.PSCmdlet](#) base class, because it provides the support needed by the Windows PowerShell runtime to expose the Windows PowerShell provider infrastructure. Note that this cmdlet also makes use of the .NET Framework regular expressions classes, such as [System.Text.RegularExpressions.Regex](#).

The following code is the class definition for this Select-Str cmdlet.

```
C#
```

```
[Cmdlet(VerbsCommon.Select, "Str",
DefaultParameterSetName="PatternParameterSet")]
```

```
public class SelectStringCommand : PSCmdlet
```

This cmdlet defines a default parameter set by adding the `DefaultParameterSetName` attribute keyword to the class declaration. The default parameter set `PatternParameterSet` is used when the `Script` parameter is not specified. For more information about this parameter set, see the `Pattern` and `Script` parameter discussion in the following section.

Defining Parameters for Data Access

This cmdlet defines several parameters that allow the user to access and examine stored data. These parameters include a `Path` parameter that indicates the location of the data store, a `Pattern` parameter that specifies the pattern to be used in the search, and several other parameters that support how the search is performed.

ⓘ Note

For more information about the basics of defining parameters, see [Adding Parameters that Process Command Line Input](#).

Declaring the Path Parameter

To locate the data store, this cmdlet must use a Windows PowerShell path to identify the Windows PowerShell provider that is designed to access the data store. Therefore, it defines a `Path` parameter of type string array to indicate the location of the provider.

C#

```
[Parameter(
    Position = 0,
    ParameterSetName = "ScriptParameterSet",
    Mandatory = true)]
[Parameter(
    Position = 0,
    ParameterSetName = "PatternParameterSet",
    ValueFromPipeline = true,
    Mandatory = true)]
    [Alias("PSPath")]
public string[] Path
{
    get { return paths; }
    set { paths = value; }
```

```
}
```

```
private string[] paths;
```

Note that this parameter belongs to two different parameter sets and that it has an alias.

Two [System.Management.Automation.ParameterAttribute](#) attributes declare that the `Path` parameter belongs to the `ScriptParameterSet` and the `PatternParameterSet`. For more information about parameter sets, see [Adding Parameter Sets to a Cmdlet](#).

The [System.Management.Automation.AliasAttribute](#) attribute declares a `PSPath` alias for the `Path` parameter. Declaring this alias is strongly recommended for consistency with other cmdlets that access Windows PowerShell providers. For more information about Windows PowerShell paths, see "PowerShell Path Concepts" in [How Windows PowerShell Works](#).

Declaring the Pattern Parameter

To specify the patterns to search for, this cmdlet declares a `Pattern` parameter that is an array of strings. A positive result is returned when any of the patterns are found in the data store. Note that these patterns can be compiled into an array of compiled regular expressions or an array of wildcard patterns used for literal searches.

```
C#
```

```
[Parameter(
    Position = 1,
    ParameterSetName = "PatternParameterSet",
    Mandatory = true)]
public string[] Pattern
{
    get { return patterns; }
    set { patterns = value; }
}
private string[] patterns;
private Regex[] regexPattern;
private WildcardPattern[] wildcardPattern;
```

When this parameter is specified, the cmdlet uses the default parameter set `PatternParameterSet`. In this case, the cmdlet uses the patterns specified here to select strings. In contrast, the `Script` parameter could also be used to provide a script that contains the patterns. The `Script` and `Pattern` parameters define two separate parameter sets, so they are mutually exclusive.

Declaring Search Support Parameters

This cmdlet defines the following support parameters that can be used to modify the search capabilities of the cmdlet.

The `Script` parameter specifies a script block that can be used to provide an alternate search mechanism for the cmdlet. The script must contain the patterns used for matching and return a `System.Management.Automation.PSObject` object. Note that this parameter is also the unique parameter that identifies the `ScriptParameterSet` parameter set. When the Windows PowerShell runtime sees this parameter, it uses only parameters that belong to the `ScriptParameterSet` parameter set.

```
C#  
  
[Parameter(  
    Position = 1,  
    ParameterSetName = "ScriptParameterSet",  
    Mandatory = true)]  
public ScriptBlock Script  
{  
    set { script = value; }  
    get { return script; }  
}  
ScriptBlock script;
```

The `SimpleMatch` parameter is a switch parameter that indicates whether the cmdlet is to explicitly match the patterns as they are supplied. When the user specifies the parameter at the command line (`true`), the cmdlet uses the patterns as they are supplied. If the parameter is not specified (`false`), the cmdlet uses regular expressions. The default for this parameter is `false`.

```
C#  
  
[Parameter]  
public SwitchParameter SimpleMatch  
{  
    get { return simpleMatch; }  
    set { simpleMatch = value; }  
}  
private bool simpleMatch;
```

The `CaseSensitive` parameter is a switch parameter that indicates whether a case-sensitive search is performed. When the user specifies the parameter at the command line (`true`), the cmdlet checks for the uppercase and lowercase of characters when comparing patterns. If the parameter is not specified (`false`), the cmdlet does not distinguish between uppercase and lowercase. For example "MyFile" and "myfile" would both be returned as positive hits. The default for this parameter is `false`.

C#

```
[Parameter]
public SwitchParameter CaseSensitive
{
    get { return caseSensitive; }
    set { caseSensitive = value; }
}
private bool caseSensitive;
```

The `Exclude` and `Include` parameters identify items that are explicitly excluded from or included in the search. By default, the cmdlet will search all items in the data store. However, to limit the search performed by the cmdlet, these parameters can be used to explicitly indicate items to be included in the search or omitted.

C#

```
[Parameter]
public SwitchParameter CaseSensitive
{
    get { return caseSensitive; }
    set { caseSensitive = value; }
}
private bool caseSensitive;
```

C#

```
[Parameter]
[ValidateNotNullOrEmpty]
public string[] Include
{
    get
    {
        return includeStrings;
    }
    set
    {
        includeStrings = value;

        this.include = new WildcardPattern[includeStrings.Length];
        for (int i = 0; i < includeStrings.Length; i++)
        {
            this.include[i] = new WildcardPattern(includeStrings[i],
WildcardOptions.IgnoreCase);
        }
    }
}
```

```
internal string[] includeStrings = null;
internal WildcardPattern[] include = null;
```

Declaring Parameter Sets

This cmdlet uses two parameter sets (`ScriptParameterSet` and `PatternParameterSet`, which is the default) as the names of two parameter sets used in data access.

`PatternParameterSet` is the default parameter set and is used when the `Pattern` parameter is specified. `ScriptParameterSet` is used when the user specifies an alternate search mechanism through the `Script` parameter. For more information about parameter sets, see [Adding Parameter Sets to a Cmdlet](#).

Overriding Input Processing Methods

Cmdlets must override one or more of the input processing methods for the `System.Management.Automation.PSCmdlet` class. For more information about the input processing methods, see [Creating Your First Cmdlet](#).

This cmdlet overrides the `System.Management.Automation.Cmdlet.BeginProcessing` method to build an array of compiled regular expressions at startup. This increases performance during searches that do not use simple matching.

C#

```
protected override void BeginProcessing()
{
    WriteDebug("Validating patterns.");
    if (patterns != null)
    {
        foreach(string pattern in patterns)
        {
            if (pattern == null)
                ThrowTerminatingError(new ErrorRecord(
                    new ArgumentNullException(
                        "Search pattern cannot be null."),
                    "NullSearchPattern",
                    ErrorCategory.InvalidArgument,
                    pattern));
        }
    }

    WriteVerbose("Search pattern(s) are valid.");

    // If a simple match is not specified, then
    // compile the regular expressions once.
    if (!simpleMatch)
```

```

{
    WriteDebug("Compiling search regular expressions.");

    RegexOptions regexOptions = RegexOptions.Compiled;
    if (!caseSensitive)
        regexOptions |= RegexOptions.Compiled;
    regexPattern = new Regex[patterns.Length];

    for (int i = 0; i < patterns.Length; i++)
    {
        try
        {
            regexPattern[i] = new Regex(patterns[i], regexOptions);
        }
        catch (ArgumentException ex)
        {
            ThrowTerminatingError(new ErrorRecord(
                ex,
                "InvalidRegularExpression",
                ErrorCategory.InvalidArgument,
                patterns[i]
            ));
        }
    }
} //Loop through patterns to create RegEx objects.

WriteVerbose("Pattern(s) compiled into regular expressions.");
}// If not a simple match.

// If a simple match is specified, then compile the
// wildcard patterns once.
else
{
    WriteDebug("Compiling search wildcards.");

    WildcardOptions wildcardOptions = WildcardOptions.Compiled;

    if (!caseSensitive)
    {
        wildcardOptions |= WildcardOptions.IgnoreCase;
    }

    wildcardPattern = new WildcardPattern[patterns.Length];
    for (int i = 0; i < patterns.Length; i++)
    {
        wildcardPattern[i] =
            new WildcardPattern(patterns[i], wildcardOptions);
    }

    WriteVerbose("Pattern(s) compiled into wildcard expressions.");
} // If match is a simple match.
}// If valid patterns are available.
}// End of function BeginProcessing().

```

This cmdlet also overrides the `System.Management.Automation.Cmdlet.ProcessRecord` method to process the string selections that the user makes on the command line. It writes the results of string selection in the form of a custom object by calling a private `MatchString` method.

```
C#  
  
protected override void ProcessRecord()  
{  
    UInt64 lineNumber = 0;  
    MatchInfo result;  
    ArrayList nonMatches = new ArrayList();  
  
    // Walk the list of paths and search the contents for  
    // any of the specified patterns.  
    foreach (string psPath in paths)  
    {  
        // Once the filepaths are expanded, we may have more than one  
        // path, so process all referenced paths.  
        foreach(PathInfo path in  
            SessionState.Path.GetResolvedPSPPathFromPSPPath(psPath)  
        )  
        {  
            WriteVerbose("Processing path " + path.Path);  
  
            // Check if the path represents one of the items to be  
            // excluded. If so, continue to next path.  
            if (!MeetsIncludeExcludeCriteria(path.ProviderPath))  
                continue;  
  
            // Get the content reader for the item(s) at the  
            // specified path.  
            Collection<IContentReader> readerCollection = null;  
            try  
            {  
                readerCollection =  
                    this.InvokeProvider.Content.GetReader(path.Path);  
            }  
            catch (PSNotSupportedException ex)  
            {  
                WriteError(new ErrorRecord(ex,  
                    "ContentAccessNotSupported",  
                    ErrorCategory.NotImplemented,  
                    path.Path)  
            );  
            return;  
        }  
  
        foreach(IContentReader reader in readerCollection)  
        {  
            // Reset the line number for this path.  
            lineNumber = 0;
```

```

// Read in a single block (line in case of a file)
// from the object.
IList items = reader.Read(1);

// Read and process one block(line) at a time until
// no more blocks(lines) exist.
while (items != null && items.Count == 1)
{
    // Increment the line number each time a line is
    // processed.
    lineNumber++;

    String message = String.Format("Testing line {0} : {1}",
                                    lineNumber, items[0]));

    WriteDebug(message);

    result = SelectString(items[0]);

    if (result != null)
    {
        result.Path = path.Path;
        result.LineNumber = lineNumber;

        WriteObject(result);
    }
    else
    {
        // Add the block(line) that did not match to the
        // collection of non matches , which will be stored
        // in the SessionState variable $NonMatches
        nonMatches.Add(items[0]);
    }

    // Get the next line from the object.
    items = reader.Read(1);

} // While loop for reading one line at a time.
}// Foreach loop for reader collection.
}// Foreach loop for processing referenced paths.
}// Foreach loop for walking of path list.

// Store the list of non-matches in the
// session state variable $NonMatches.
try
{
    this.SessionState.PSVariable.Set("NonMatches", nonMatches);
}
catch (SessionStateUnauthorizedAccessException ex)
{
    WriteError(new ErrorRecord(ex,
                               "CannotWriteVariableNonMatches",
                               ErrorCategory.InvalidOperation,
                               nonMatches));
}

```

```
}

} // End of protected override void ProcessRecord().
```

Accessing Content

Your cmdlet must open the provider indicated by the Windows PowerShell path so that it can access the data. The [System.Management.Automation.SessionState](#) object for the runspace is used for access to the provider, while the [System.Management.Automation.PSCmdlet.InvokeProvider*](#) property of the cmdlet is used to open the provider. Access to content is provided by retrieval of the [System.Management.Automation.ProviderIntrinsics](#) object for the provider opened.

This sample Select-Str cmdlet uses the [System.Management.Automation.ProviderIntrinsics.Content*](#) property to expose the content to scan. It can then call the [System.Management.Automation.ContentCmdletProviderIntrinsics.GetReader*](#) method, passing the required Windows PowerShell path.

Code Sample

The following code shows the implementation of this version of this Select-Str cmdlet. Note that this code includes the cmdlet class, private methods used by the cmdlet, and the Windows PowerShell snap-in code used to register the cmdlet. For more information about registering the cmdlet, see [Building the Cmdlet](#).

C#

```
//
// Copyright (c) 2006 Microsoft Corporation. All rights reserved.
//
// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
// PARTICULAR PURPOSE.
//
using System;
using System.Text.RegularExpressions;
using System.Collections;
using System.Collections.ObjectModel;
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.ComponentModel;

namespace Microsoft.Samples.PowerShell.Commands
```

```

#region SelectStringCommand
/// <summary>
/// This cmdlet searches through PSObjects for particular patterns.
/// </summary>
/// <remarks>
/// This cmdlet can be used to search any object, such as a file or a
/// variable, whose provider exposes methods for reading and writing
/// content.
/// </remarks>
[Cmdlet(VerbsCommon.Select, "Str",
DefaultParameterSetName="PatternParameterSet")]
public class SelectStringCommand : PSCmdlet
{
    #region Parameters
    /// <summary>
    /// Declare a Path parameter that specifies where the data is stored.
    /// This parameter must specify a PowerShell that indicates the
    /// PowerShell provider that is used to access the objects to be
    /// searched for matching patterns. This parameter should also have
    /// a PSPath alias to provide consistency with other cmdlets that use
    /// PowerShell providers.
    /// </summary>
    /// <value>Path of the object(s) to search.</value>
    [Parameter(
        Position = 0,
        ParameterSetName = "ScriptParameterSet",
        Mandatory = true)]
    [Parameter(
        Position = 0,
        ParameterSetName = "PatternParameterSet",
        ValueFromPipeline = true,
        Mandatory = true)]
        [Alias("PSPath")]
    public string[] Path
    {
        get { return paths; }
        set { paths = value; }
    }
    private string[] paths;

    /// <summary>
    /// Declare a Pattern parameter that specifies the pattern(s)
    /// used to find matching patterns in the string representation
    /// of the objects. A positive result will be returned
    /// if any of the patterns are found in the objects.
    /// </summary>
    /// <remarks>
    /// The patterns will be compiled into an array of wildcard
    /// patterns for a simple match (literal string matching),
    /// or the patterns will be converted into an array of compiled
    /// regular expressions.
    /// </remarks>
    /// <value>Array of patterns to search.</value>
    [Parameter(

```

```

        Position = 1,
        ParameterSetName = "PatternParameterSet",
        Mandatory = true)]
public string[] Pattern
{
    get { return patterns; }
    set { patterns = value; }
}
private string[] patterns;
private Regex[] regexPattern;
private WildcardPattern[] wildcardPattern;

/// <summary>
/// Declare a Script parameter that specifies a script block
/// that is called to perform the matching operations
/// instead of the matching performed by the cmdlet.
/// </summary>
/// <value>Script block that will be called for matching</value>
[Parameter(
    Position = 1,
    ParameterSetName = "ScriptParameterSet",
    Mandatory = true)]
public ScriptBlock Script
{
    set { script = value; }
    get { return script; }
}
ScriptBlock script;

/// <summary>
/// Declare a switch parameter that specifies if the pattern(s) are used
/// literally. If not (default), searching is
/// done using regular expressions.
/// </summary>
/// <value>If True, a literal pattern is used.</value>
[Parameter]
public SwitchParameter SimpleMatch
{
    get { return simpleMatch; }
    set { simpleMatch = value; }
}
private bool simpleMatch;

/// <summary>
/// Declare a switch parameter that specifies if a case-sensitive
/// search is performed. If not (default), a case-insensitive search
/// is performed.
/// </summary>
/// <value>If True, a case-sensitive search is made.</value>
[Parameter]
public SwitchParameter CaseSensitive
{
    get { return caseSensitive; }
    set { caseSensitive = value; }
}

```

```
private bool caseSensitive;

/// <summary>
/// Declare an Include parameter that species which
/// specific items are searched. When this parameter
/// is used, items that are not listed here are omitted
/// from the search.
/// </summary>
[Parameter]
[ValidateNotNullOrEmpty]
public string[] Include
{
    get
    {
        return includeStrings;
    }
    set
    {
        includeStrings = value;

        this.include = new WildcardPattern[includeStrings.Length];
        for (int i = 0; i < includeStrings.Length; i++)
        {
            this.include[i] = new WildcardPattern(includeStrings[i],
WildcardOptions.IgnoreCase);
        }
    }
}

internal string[] includeStrings = null;
internal WildcardPattern[] include = null;

/// <summary>
/// Declare an Exclude parameter that species which
/// specific items are omitted from the search.
/// </summary>
///
[Parameter]
[ValidateNotNullOrEmpty]
public string[] Exclude
{
    get
    {
        return excludeStrings;
    }
    set
    {
        excludeStrings = value;

        this.exclude = new WildcardPattern[excludeStrings.Length];
        for (int i = 0; i < excludeStrings.Length; i++)
        {
            this.exclude[i] = new WildcardPattern(excludeStrings[i],
WildcardOptions.IgnoreCase);
        }
    }
}
```

```

        }

    }

    internal string[] excludeStrings;
    internal WildcardPattern[] exclude;

#endregion Parameters

#region Overrides
/// <summary>
/// If regular expressions are used for pattern matching,
/// then build an array of compiled regular expressions
/// at startup. This increases performance during scanning
/// operations when simple matching is not used.
/// </summary>
protected override void BeginProcessing()
{
    WriteDebug("Validating patterns.");
    if (patterns != null)
    {
        foreach(string pattern in patterns)
        {
            if (pattern == null)
                ThrowTerminatingError(new ErrorRecord(
                    new ArgumentNullException(
                        "Search pattern cannot be null."),
                    "NullSearchPattern",
                    ErrorCategory.InvalidArgument,
                    pattern)
                );
        }

        WriteVerbose("Search pattern(s) are valid.");
    }

    // If a simple match is not specified, then
    // compile the regular expressions once.
    if (!simpleMatch)
    {
        WriteDebug("Compiling search regular expressions.");

        RegexOptions regexOptions = RegexOptions.Compiled;
        if (!caseSensitive)
            regexOptions |= RegexOptions.Compiled;
        regexPattern = new Regex[patterns.Length];

        for (int i = 0; i < patterns.Length; i++)
        {
            try
            {
                regexPattern[i] = new Regex(patterns[i], regexOptions);
            }
            catch (ArgumentException ex)
            {
                ThrowTerminatingError(new ErrorRecord(
                    ex,
                    "InvalidRegularExpression",

```

```

                ErrorCategory.InvalidArgument,
                patterns[i]
            )));
        }
    } //Loop through patterns to create RegEx objects.

    WriteVerbose("Pattern(s) compiled into regular expressions.");
} // If not a simple match.

// If a simple match is specified, then compile the
// wildcard patterns once.
else
{
    WriteDebug("Compiling search wildcards.");

    WildcardOptions wildcardOptions = WildcardOptions.Compiled;

    if (!caseSensitive)
    {
        wildcardOptions |= WildcardOptions.IgnoreCase;
    }

    wildcardPattern = new WildcardPattern[patterns.Length];
    for (int i = 0; i < patterns.Length; i++)
    {
        wildcardPattern[i] =
            new WildcardPattern(patterns[i], wildcardOptions);
    }

    WriteVerbose("Pattern(s) compiled into wildcard expressions.");
} // If match is a simple match.
} // If valid patterns are available.
} // End of function BeginProcessing().

/// <summary>
/// Process the input and search for the specified patterns.
/// </summary>
protected override void ProcessRecord()
{
    UInt64 lineNumber = 0;
    MatchInfo result;
    ArrayList nonMatches = new ArrayList();

    // Walk the list of paths and search the contents for
    // any of the specified patterns.
    foreach (string psPath in paths)
    {
        // Once the filepaths are expanded, we may have more than one
        // path, so process all referenced paths.
        foreach(PathInfo path in
            SessionState.Path.GetResolvedPSPPathFromPSPPath(psPath)
        )
        {
            WriteVerbose("Processing path " + path.Path);

```

```

// Check if the path represents one of the items to be
// excluded. If so, continue to next path.
if (!MeetsIncludeExcludeCriteria(path.ProviderPath))
    continue;

// Get the content reader for the item(s) at the
// specified path.
Collection<IContentReader> readerCollection = null;
try
{
    readerCollection =
        this.InvokeProvider.Content.GetReader(path.Path);
}
catch (PSNotSupportedException ex)
{
    WriteError(new ErrorRecord(ex,
        "ContentAccessNotSupported",
        ErrorCategory.NotImplemented,
        path.Path)
    );
    return;
}

foreach(IContentReader reader in readerCollection)
{
    // Reset the line number for this path.
    lineNumber = 0;

    // Read in a single block (line in case of a file)
    // from the object.
    IList items = reader.Read(1);

    // Read and process one block(line) at a time until
    // no more blocks(lines) exist.
    while (items != null && items.Count == 1)
    {
        // Increment the line number each time a line is
        // processed.
        lineNumber++;

        String message = String.Format("Testing line {0} : {1}",
            lineNumber, items[0]);

        WriteDebug(message);

        result = SelectString(items[0]);

        if (result != null)
        {
            result.Path = path.Path;
            result.LineNumber = lineNumber;

            WriteObject(result);
        }
        else

```

```

    {
        // Add the block(line) that did not match to the
        // collection of non matches , which will be stored
        // in the SessionState variable $NonMatches
        nonMatches.Add(items[0]);
    }

    // Get the next line from the object.
    items = reader.Read(1);

    } // While loop for reading one line at a time.
    } // Foreach loop for reader collection.
    } // Foreach loop for processing referenced paths.
} // Foreach loop for walking of path list.

// Store the list of non-matches in the
// session state variable $NonMatches.
try
{
    this.SessionState.PSVariable.Set("NonMatches", nonMatches);
}
catch (SessionStateUnauthorizedAccessException ex)
{
    WriteError(new ErrorRecord(ex,
        "CannotWriteVariableNonMatches",
        ErrorCategory.InvalidOperation,
        nonMatches)
    );
}

} // End of protected override void ProcessRecord().
#endregion Overrides

#region PrivateMethods
/// <summary>
/// Check for a match using the input string and the pattern(s)
/// specified.
/// </summary>
/// <param name="input">The string to test.</param>
/// <returns>MatchInfo object containing information about
/// result of a match</returns>
private MatchInfo SelectString(object input)
{
    string line = null;

    try
    {
        // Convert the object to a string type
        // safely using language support methods
        line = (string)LanguagePrimitives.ConvertTo(
            input,
            typeof(string)
        );
        line = line.Trim(' ', '\t');
    }
}

```

```

        catch (PSInvalidCastException ex)
    {
        WriteError(new ErrorRecord(
            ex,
            "CannotCastObjectToString",
            ErrorCategory.InvalidOperation,
            input)
        );
    }

    return null;
}

MatchInfo result = null;

// If a scriptblock has been specified, call it
// with the path for processing.  It will return
// one object.
if (script != null)
{
    WriteDebug("Executing script block.");

    Collection<PSObject> psObjects =
        script.Invoke(
            line,
            simpleMatch,
            caseSensitive
        );

    foreach (PSObject psObject in psObjects)
    {
        if (LanguagePrimitives.IsTrue(psObject))
        {
            result = new MatchInfo();
            result.Line = line;
            result.IgnoreCase = !caseSensitive;

            break;
        } //End of If.
    } //End ForEach loop.
} // End of If if script exists.

// If script block exists, see if this line matches any
// of the match patterns.
else
{
    int patternIndex = 0;

    while (patternIndex < patterns.Length)
    {
        if ((simpleMatch &&
            wildcardPattern[patternIndex].IsMatch(line))
            || (regexPattern != null
            && regexPattern[patternIndex].IsMatch(line))
        )
    }
}

```

```

        result = new MatchInfo();
        result.IgnoreCase = !caseSensitive;
        result.Line = line;
        result.Pattern = patterns[patternIndex];

        break;
    }

    patternIndex++;

}// While loop through patterns.
// Else for no script block specified.

return result;

}// End of SelectString

/// <summary>
/// Check whether the supplied name meets the include/exclude criteria.
/// That is - it's on the include list if the include list was
/// specified, and not on the exclude list if the exclude list was
specified.

/// </summary>
/// <param name="path">path to validate</param>
/// <returns>True if the path is acceptable.</returns>
private bool MeetsIncludeExcludeCriteria(string path)
{
    bool ok = false;

    // See if the file is on the include list.
    if (this.include != null)
    {
        foreach (WildcardPattern patternItem in this.include)
        {
            if (patternItem.IsMatch(path))
            {
                ok = true;
                break;
            }
        }
    }
    else
    {
        ok = true;
    }

    if (!ok)
        return false;

    // See if the file is on the exclude list.
    if (this.exclude != null)
    {
        foreach (WildcardPattern patternItem in this.exclude)
        {
            if (patternItem.IsMatch(path))

```

```

        {
            ok = false;
            break;
        }
    }

    return ok;
} //MeetsIncludeExcludeCriteria
#endregion Private Methods

}// class SelectStringCommand

#endregion SelectStringCommand

#region MatchInfo

/// <summary>
/// Class representing the result of a pattern/literal match
/// that is passed through the pipeline by the Select-Str cmdlet.
/// </summary>
public class MatchInfo
{
    /// <summary>
    /// Indicates if the match was done ignoring case.
    /// </summary>
    /// <value>True if case was ignored.</value>
    public bool IgnoreCase
    {
        get { return ignoreCase; }
        set { ignoreCase = value; }
    }
    private bool ignoreCase;

    /// <summary>
    /// Specifies the number of the matching line.
    /// </summary>
    /// <value>The number of the matching line.</value>
    public UInt64 LineNumber
    {
        get { return lineNumber; }
        set { lineNumber = value; }
    }
    private UInt64 lineNumber;

    /// <summary>
    /// Specifies the text of the matching line.
    /// </summary>
    /// <value>The text of the matching line.</value>
    public string Line
    {
        get { return line; }
        set { line = value; }
    }
    private string line;
}

```

```
/// <summary>
/// Specifies the full path of the object(file) containing the
/// matching line.
/// </summary>
/// <remarks>
/// It will be "InputStream" if the object came from the input
/// stream.
/// </remarks>
/// <value>The path name</value>
public string Path
{
    get { return path; }
    set
    {
        pathSet = true;
        path = value;
    }
}
private string path;
private bool pathSet;

/// <summary>
/// Specifies the pattern that was used in the match.
/// </summary>
/// <value>The pattern string</value>
public string Pattern
{
    get { return pattern; }
    set { pattern = value; }
}
private string pattern;

private const string MatchFormat = "{0}:{1}:{2}";

/// <summary>
/// Returns the string representation of this object. The format
/// depends on whether a path has been set for this object or
/// not.
/// </summary>
/// <remarks>
/// If the path component is set, as would be the case when
/// matching in a file, ToString() returns the path, line
/// number and line text. If path is not set, then just the
/// line text is presented.
/// </remarks>
/// <returns>The string representation of the match object.</returns>
public override string ToString()
{
    if (pathSet)
        return String.Format(
            System.Threading.Thread.CurrentThread.CurrentCulture,
            MatchFormat,
            this.path,
            this.lineNumber,
```

```
        this.line
    );
else
    return this.line;
}
}// End class MatchInfo

#endregion

#region PowerShell snap-in

/// <summary>
/// Create a PowerShell snap-in for the Select-Str cmdlet.
/// </summary>
[RunInstaller(true)]
public class SelectStringPSSnapIn : PSSnapIn
{
    /// <summary>
    /// Create an instance of the SelectStrPSSnapin class.
    /// </summary>
    public SelectStringPSSnapIn()
        : base()
    {

    }

    /// <summary>
    /// Specify the name of the PowerShell snap-in.
    /// </summary>
    public override string Name
    {
        get
        {
            return "SelectStrPSSnapIn";
        }
    }

    /// <summary>
    /// Specify the vendor of the PowerShell snap-in.
    /// </summary>
    public override string Vendor
    {
        get
        {
            return "Microsoft";
        }
    }

    /// <summary>
    /// Specify the localization resource information for the vendor.
    /// Use the format: SnapinName,VendorName.
    /// </summary>
    public override string VendorResource
    {
        get
        {
```

```

        return "SelectStrSnapIn,Microsoft";
    }

}

/// <summary>
/// Specify the description of the PowerShell snap-in.
/// </summary>
public override string Description
{
    get
    {
        return "This is a PowerShell snap-in for the Select-Str cmdlet.";
    }
}

/// <summary>
/// Specify the localization resource information for the description.
/// Use the format: SnapinName,Description.

/// </summary>
public override string DescriptionResource
{
    get
    {
        return "SelectStrSnapIn,This is a PowerShell snap-in for the
Select-Str cmdlet.";
    }
}
}

#endregion PowerShell snap-in

} //namespace Microsoft.Samples.PowerShell.Commands;

```

Building the Cmdlet

After implementing a cmdlet, you must register it with Windows PowerShell through a Windows PowerShell snap-in. For more information about registering cmdlets, see [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Cmdlet

When your cmdlet has been registered with Windows PowerShell, you can test it by running it on the command line. The following procedure can be used to test the sample Select-Str cmdlet.

1. Start Windows PowerShell, and search the Notes file for occurrences of lines with the expression ".NET". Note that the quotation marks around the name of the path are required only if the path consists of more than one word.

PowerShell

```
Select-Str -Path "notes" -Pattern ".NET" -SimpleMatch=$false
```

The following output appears.

Output

```
IgnoreCase      : True
LineNumber      : 8
Line            : Because Windows PowerShell works directly with .NET
objects, there is often a .NET object
Path            : C:\PowerShell-Progs\workspace\Samples\SelectStr\notes
Pattern         : .NET
IgnoreCase      : True
LineNumber      : 21
Line            : You should normally define the class for a cmdlet in a
.NET namespace
Path            : C:\PowerShell-Progs\workspace\Samples\SelectStr\notes
Pattern         : .NET
```

2. Search the Notes file for occurrences of lines with the word "over", followed by any other text. The `SimpleMatch` parameter is using the default value of `false`. The search is case-insensitive because the `CaseSensitive` parameter is set to `false`.

PowerShell

```
Select-Str -Path notes -Pattern "over*" -SimpleMatch -
CaseSensitive:$false
```

The following output appears.

Output

```
IgnoreCase      : True
LineNumber      : 45
Line            : Override StopProcessing
Path            : C:\PowerShell-Progs\workspace\Samples\SelectStr\notes
Pattern         : over*
IgnoreCase      : True
LineNumber      : 49
Line            : overriding the StopProcessing method
Path            : C:\PowerShell-Progs\workspace\Samples\SelectStr\notes
Pattern         : over*
```

3. Search the Notes file using a regular expression as the pattern. The cmdlet searches for alphabetical characters and blank spaces enclosed in parentheses.

PowerShell

```
Select-Str -Path notes -Pattern "\([A-Za-z:blank:]" -SimpleMatch:$false
```

The following output appears.

Output

```
IgnoreCase      : True
LineNumber     : 1
Line           : Advisory Guidelines (Consider Following)
Path           : C:\PowerShell-Progs\workspace\Samples\SelectStr\notes
Pattern        : \([A-Za-z:blank:]
IgnoreCase      : True
LineNumber     : 53
Line           : If your cmdlet has objects that are not disposed of
(written to the pipeline)
Path           : C:\PowerShell-Progs\workspace\Samples\SelectStr\notes
Pattern        : \([A-Za-z:blank:]
```

4. Perform a case-sensitive search of the Notes file for occurrences of the word "Parameter".

PowerShell

```
Select-Str -Path notes -Pattern Parameter -CaseSensitive
```

The following output appears.

Output

```
IgnoreCase      : False
LineNumber     : 6
Line           : Support an InputObject Parameter
Path           : C:\PowerShell-Progs\workspace\Samples\SelectStr\notes
Pattern        : Parameter
IgnoreCase      : False
LineNumber     : 30
Line           : Support Force Parameter
Path           : C:\PowerShell-Progs\workspace\Samples\SelectStr\notes
Pattern        : Parameter
```

5. Search the Variable provider shipped with Windows PowerShell for variables that have numerical values from 0 through 9.

PowerShell

```
Select-Str -Path * -Pattern "[0-9]"
```

The following output appears.

Output

```
IgnoreCase      : True
LineNumber      : 1
Line            : 64
Path            : Variable:\MaximumHistoryCount
Pattern         : [0-9]
```

6. Use a script block to search the file `SelectStrCommandSample.cs` for the string "Pos". The `-cmatch` operator performs a case-insensitive pattern match.

PowerShell

```
Select-Str -Path "SelectStrCommandSample.cs" -Script { if ($args[0] -cmatch "Pos"){ return $true } return $false }
```

The following output appears.

Output

```
IgnoreCase      : True
LineNumber      : 37
Line            : Position = 0.
Path            : C:\PowerShell-
Progs\workspace\Samples\SelectStr\SelectStrCommandSample.cs
Pattern         :
```

See Also

[How to Create a Windows PowerShell Cmdlet](#)

[Creating Your First Cmdlet](#)

[Creating a Cmdlet that Modifies the System](#)

[Design Your Windows PowerShell Provider](#)

[How Windows PowerShell Works ↗](#)

[How to Register Cmdlets, Providers, and Host Applications ↗](#)

Windows PowerShell SDK

Cmdlet Samples

Article • 09/15/2023

This section describes sample code that is provided in the Windows PowerShell 2.0 SDK.

In This Section

- [GetProcessSample01 Sample](#): This sample shows how to write a cmdlet that retrieves the processes on the local computer.
- [GetProcessSample02 Sample](#): This sample shows how to write a cmdlet that retrieves the processes on the local computer. It provides a Name parameter that can be used to specify the processes to be retrieved.
- [GetProcessSample03 Sample](#): This sample shows how to write a cmdlet that retrieves the processes on the local computer. It provides a Name parameter that can accept an object from the pipeline or a value from a property of an object whose property name is the same as the parameter name.
- [GetProcessSample04 Sample](#): This sample shows how to write a cmdlet that retrieves the processes on the local computer. It generates a non-terminating error if an error occurs while retrieving a process.
- [GetProcessSample05 Sample](#): This sample shows a complete version of the Get-Proc cmdlet.
- [StopProcessSample01 Sample](#): This sample shows how to write a cmdlet that requests feedback from the user before it attempts to stop a process, and how to implement a `PassThru` parameter that indicates that the user wants the cmdlet to return an object.
- [StopProcessSample02 Sample](#): This sample shows how to write a cmdlet that writes debug, verbose, and warning messages while stopping processes on the local computer.
- [StopProcessSample03 Sample](#): This sample shows how to write a cmdlet whose parameters have aliases and that support wildcard characters.
- [StopProcessSample04 Sample](#): This sample shows how to write a cmdlet that declares parameter sets, specifies the default parameter set, and can accept an input object.

- **Events01 Sample:** This sample shows how to create a cmdlet that allows the user to register for events raised by `System.IO.FileSystemWatcher`. With this cmdlet users can, for example, register an action to execute when a file is created under a specific directory. This sample derives from the `Microsoft.PowerShell.Commands.ObjectEventRegistrationBase` base class.

See Also

- [Writing a Windows PowerShell Cmdlet](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

GetProcessSample01 Sample

Article • 04/10/2024

This sample shows how to implement a cmdlet that retrieves the processes on the local computer. This cmdlet is a simplified version of the `Get-Process` cmdlet that is provided by Windows PowerShell 2.0.

How to build the sample by using Visual Studio

1. With the Windows PowerShell 2.0 SDK installed, navigate to the GetProcessSample01 folder. The default location is `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0\Samples\sysmgmt\WindowsPowerShell\csharp\GetProcessSample01`.
2. Double-click the icon for the solution (.sln) file. This opens the sample project in Microsoft Visual Studio.
3. In the **Build** menu, select **Build Solution** to build the library for the sample in the default `\bin` or `\bin\debug` folders.

How to run the sample

1. Open a Command Prompt window.
2. Navigate to the directory containing the sample .dll file.
3. Run `installutil "GetProcessSample01.dll"`.
4. Start Windows PowerShell.
5. Run the following command to add the snap-in to the shell.

```
Add-PSSnapin GetProcPSSnapIn01
```

6. Enter the following command to run the cmdlet. `Get-Proc`

```
Get-Proc
```

This is a sample output that results from following these steps.

```
Output
```

Id	Name	State	HasMoreData	Location
--	-----	-----	-----	-----

1	26932870-d3b...	NotStarted	False	

```
Write-Host "A f..."
```

PowerShell

```
Set-Content $Env:TEMP\test.txt "This is a test file"
```

Output

```
A file was created in the TEMP directory
```

Requirements

This sample requires Windows PowerShell 1.0 or later.

Demonstrates

This sample demonstrates the following.

- Creating a basic sample cmdlet.
- Defining a cmdlet class by using the Cmdlet attribute.
- Creating a snap-in that works with both Windows PowerShell 1.0 and Windows PowerShell 2.0. Subsequent samples use modules instead of snap-ins so they require Windows PowerShell 2.0.

Example

This sample shows how to create a simple cmdlet and its snap-in.

C#

```
using System;
using System.Diagnostics;
using System.Management.Automation;           //Windows PowerShell
namespace
using System.ComponentModel;
```

```
namespace Microsoft.Samples.PowerShell.Commands
{

    #region GetProcCommand

        /// <summary>
        /// This class implements the Get-Proc cmdlet.
        /// </summary>
        [Cmdlet(VerbsCommon.Get, "Proc")]
        public class GetProcCommand : Cmdlet
    {
        #region Cmdlet Overrides

            /// <summary>
            /// The ProcessRecord method calls the Process.GetProcesses
            /// method to retrieve the processes of the local computer.
            /// Then, the WriteObject method writes the associated processes
            /// to the pipeline.
            /// </summary>
            protected override void ProcessRecord()
        {
            // Retrieve the current processes.
            Process[] processes = Process.GetProcesses();

            // Write the processes to the pipeline to make them available
            // to the next cmdlet. The second argument (true) tells Windows
            // PowerShell to enumerate the array and to send one process
            // object at a time to the pipeline.
            WriteObject(processes, true);
        }

        #endregion Overrides
    } //GetProcCommand

    #endregion GetProcCommand

    #region PowerShell snap-in

        /// <summary>
        /// Create this sample as a PowerShell snap-in
        /// </summary>
        [RunInstaller(true)]
        public class GetProcPSSnapIn01 : PSSnapIn
    {
        #region <summary>
        /// Create an instance of the GetProcPSSnapIn01
        #endregion
        public GetProcPSSnapIn01()
            : base()
        {

        }

        #region <summary>

```

```

    /// Get a name for this PowerShell snap-in. This name will be used in
registering
    /// this PowerShell snap-in.
    /// </summary>
    public override string Name
    {
        get
        {
            return "GetProcPSSnapIn01";
        }
    }

    /// <summary>
    /// Vendor information for this PowerShell snap-in.
    /// </summary>
    public override string Vendor
    {
        get
        {
            return "Microsoft";
        }
    }

    /// <summary>
    /// Gets resource information for vendor. This is a string of format:
    /// resourceBaseName,resourceName.
    /// </summary>
    public override string VendorResource
    {
        get
        {
            return "GetProcPSSnapIn01,Microsoft";
        }
    }

    /// <summary>
    /// Description of this PowerShell snap-in.
    /// </summary>
    public override string Description
    {
        get
        {
            return "This is a PowerShell snap-in that includes the Get-
Proc cmdlet.";
        }
    }
}

#endregion PowerShell snap-in
}

```

See Also

- Writing a Windows PowerShell Cmdlet

GetProcessSample02 Sample

Article • 09/15/2023

This sample shows how to write a cmdlet that retrieves the processes on the local computer. It provides a `Name` parameter that can be used to specify the processes to be retrieved. This cmdlet is a simplified version of the `Get-Process` cmdlet provided by Windows PowerShell 2.0.

How to build the sample using Visual Studio

1. With the Windows PowerShell 2.0 SDK installed, navigate to the GetProcessSample02 folder. The default location is `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0\Samples\sysmgmt\WindowsPowerShell\csharp\GetProcessSample02.`
2. Double-click the icon for the solution (.sln) file. This opens the sample project in Visual Studio.
3. In the **Build** menu, select **Build Solution** to build the library for the sample in the default `\bin` or `\bin\debug` folders.

How to run the sample

1. Create the following module folder:
`[user]\Documents\WindowsPowerShell\Modules\GetProcessSample02`
2. Copy the sample assembly to the module folder.
3. Start Windows PowerShell.
4. Run the following command to load the assembly into Windows PowerShell:
`Import-Module getprocesssample02`
5. Run the following command to run the cmdlet:
`Get-Proc`

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Declaring a cmdlet class using the Cmdlet attribute.
- Declaring a cmdlet parameter using the Parameter attribute.
- Specifying the position of the parameter.
- Declaring a validation attribute for the parameter input.

Example

This sample shows an implementation of the Get-Proc cmdlet that includes a `Name` parameter.

C#

```
namespace Microsoft.Samples.PowerShell.Commands
{
    using System;
    using System.Diagnostics;
    using System.Management.Automation; // Windows PowerShell namespace

    #region GetProcCommand

    /// <summary>
    /// This class implements the Get-Proc cmdlet.
    /// </summary>
    [Cmdlet(VerbsCommon.Get, "Proc")]
    public class GetProcCommand : Cmdlet
    {
        #region Parameters

        /// <summary>
        /// The names of the processes retrieved by the cmdlet.
        /// </summary>
        private string[] processNames;

        /// <summary>
        /// Gets or sets the list of process names on which
        /// the Get-Proc cmdlet will work.
        /// </summary>
        [Parameter(Position = 0)]
        [ValidateNotNullOrEmpty]
        public string[] Name
```

```

{
    get { return this.processNames; }
    set { this.processNames = value; }
}

#endregion Parameters

#region Cmdlet Overrides

/// <summary>
/// The ProcessRecord method calls the Process.GetProcesses
/// method to retrieve the processes specified by the Name
/// parameter. Then, the WriteObject method writes the
/// associated process objects to the pipeline.
/// </summary>
protected override void ProcessRecord()
{
    // If no process names are passed to the cmdlet, get all
    // processes.
    if (this.processNames == null)
    {
        WriteObject(Process.GetProcesses(), true);
    }
    else
    {
        // If process names are passed to cmdlet, get and write
        // the associated processes.
        foreach (string name in this.processNames)
        {
            WriteObject(Process.GetProcessesByName(name), true);
        }
    } // End if (processNames...).
} // End ProcessRecord.
#endregion Cmdlet Overrides
} // End GetProcCommand class.
#endregion GetProcCommand
}

```

See Also

- [Writing a Windows PowerShell Cmdlet](#)

GetProcessSample03 Sample

Article • 03/24/2025

This sample shows how to implement a cmdlet that retrieves the processes on the local computer. It provides a `Name` parameter that can accept an object from the pipeline or a value from a property of an object whose property name is the same as the parameter name. This cmdlet is a simplified version of the `Get-Process` cmdlet provided by Windows PowerShell 2.0.

How to build the sample using Visual Studio

1. With the Windows PowerShell 2.0 SDK installed, navigate to the GetProcessSample03 folder. The default location is `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0\Samples\sysmgmt\WindowsPowerShell\csharp\GetProcessSample03`.
2. Double-click the icon for the solution (.sln) file. This opens the sample project in Visual Studio.
3. In the **Build** menu, select **Build Solution** to build the library for the sample in the default `\bin` or `\bin\debug` folders.

How to run the sample

1. Create the following module folder:
`[user]\Documents\WindowsPowerShell\Modules\GetProcessSample03`
2. Copy the sample assembly to the module folder.
3. Start Windows PowerShell.
4. Run the following command to load the assembly into Windows PowerShell:
`Import-Module getprocesssample03`
5. Run the following command to run the cmdlet:
`Get-Proc`

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Declaring a cmdlet class using the Cmdlet attribute.
- Declaring a cmdlet parameter using the Parameter attribute.
- Specifying the position of the parameter.
- Specifying that the parameter takes input from the pipeline. The input can be taken from an object or a value from a property of an object whose property name is the same as the parameter name.
- Declaring a validation attribute for the parameter input.

Example

This sample shows an implementation of the Get-Proc cmdlet that includes a `Name` parameter that accepts input from the pipeline.

C#

```
namespace Microsoft.Samples.PowerShell.Commands
{
    using System;
    using System.Diagnostics;
    using System.Management.Automation; // Windows PowerShell
    namespace
        #region GetProcCommand

        /// <summary>
        /// This class implements the Get-Proc cmdlet.
        /// </summary>
        [Cmdlet(VerbsCommon.Get, "Proc")]
        public class GetProcCommand : Cmdlet
        {
            #region Parameters

            /// <summary>
            /// The names of the processes retrieved by the cmdlet.
            /// </summary>
            private string[] processNames;

            /// <summary>
            /// Gets or sets the names of the
        }
}
```

```

/// process that the cmdlet will retrieve.
/// </summary>
[Parameter(
    Position = 0,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true)]
[ValidateNotNullOrEmpty]
public string[] Name
{
    get { return this.processNames; }
    set { this.processNames = value; }
}

#endregion Parameters

#region Cmdlet Overrides

/// <summary>
/// The ProcessRecord method calls the Process.GetProcesses
/// method to retrieve the processes specified by the Name
/// parameter. Then, the WriteObject method writes the
/// associated processes to the pipeline.
/// </summary>
protected override void ProcessRecord()
{
    // If no process names are passed to the cmdlet, get all
    // processes.
    if (this.processNames == null)
    {
        WriteObject(Process.GetProcesses(), true);
    }
    else
    {
        // If process names are passed to the cmdlet, get and write
        // the associated processes.
        foreach (string name in this.processNames)
        {
            WriteObject(Process.GetProcessesByName(name), true);
        }
    } // End if (processNames ...)
} // End ProcessRecord.

#endregion Overrides
} // End GetProcCommand.
#endregion GetProcCommand
}

```

See Also

- [Writing a Windows PowerShell Cmdlet](#)

GetProcessSample04 Sample

Article • 03/24/2025

This sample shows how to implement a cmdlet that retrieves the processes on the local computer. It generates a non-terminating error if an error occurs while retrieving a process. This cmdlet is a simplified version of the `Get-Process` cmdlet provided by Windows PowerShell 2.0.

How to build the sample using Visual Studio

1. With the Windows PowerShell 2.0 SDK installed, navigate to the GetProcessSample04 folder. The default location is `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0\Samples\sysmgmt\WindowsPowerShell\csharp\GetProcessSample04`.
2. Double-click the icon for the solution (.sln) file. This opens the sample project in Visual Studio.
3. In the **Build** menu, select **Build Solution** to build the library for the sample in the default `\bin` or `\bin\debug` folders.

How to run the sample

1. Create the following module folder:

```
[user]\Documents\WindowsPowerShell\Modules\GetProcessSample04
```

2. Copy the sample assembly to the module folder.
3. Start Windows PowerShell.
4. Run the following command to load the assembly into Windows PowerShell:

```
Import-Module getprocesssample04
```

5. Run the following command to run the cmdlet:

```
Get-Proc
```

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Declaring a cmdlet class using the Cmdlet attribute.
- Declaring a cmdlet parameter using the Parameter attribute.
- Specifying the position of the parameter.
- Specifying that the parameter takes input from the pipeline. The input can be taken from an object or a value from a property of an object whose property name is the same as the parameter name.
- Declaring a validation attribute for the parameter input.
- Trapping a non-terminating error and writing an error message to the error stream.

Example

This sample shows how to create a cmdlet that handles non-terminating errors and writes error messages to the error stream.

C#

```
namespace Microsoft.Samples.PowerShell.Commands
{
    using System;
    using System.Diagnostics;
    using System.Management.Automation;      // Windows PowerShell
namespace.
    #region GetProcCommand

    /// <summary>
    /// This class implements the Get-Proc cmdlet.
    /// </summary>
    [Cmdlet(VerbsCommon.Get, "Proc")]
    public class GetProcCommand : Cmdlet
    {
        #region Parameters

        /// <summary>
        /// The names of the processes to act on.
        /// </summary>
        private string[] processNames;

        /// <summary>
        /// Gets or sets the list of process names on
        /// which the Get-Proc cmdlet will work.
    }
}
```

```

/// </summary>
[Parameter(
    Position = 0,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true)]
[ValidateNotNullOrEmpty]
public string[] Name
{
    get { return this.processNames; }
    set { this.processNames = value; }
}

#endregion Parameters

#region Cmdlet Overrides

/// <summary>
/// The ProcessRecord method calls the Process.GetProcesses
/// method to retrieve the processes specified by the Name
/// parameter. Then, the WriteObject method writes the
/// associated processes to the pipeline.
/// </summary>
protected override void ProcessRecord()
{
    // If no process names are passed to cmdlet, get all
    // processes.
    if (this.processNames == null)
    {
        WriteObject(Process.GetProcesses(), true);
    }
    else
    {
        // If process names are passed to the cmdlet, get and write
        // the associated processes.
        // If a non-terminating error occurs while retrieving
processes,
        // call the WriteError method to send an error record to the
        // error stream.
        foreach (string name in this.processNames)
        {
            Process[] processes;

            try
            {
                processes = Process.GetProcessesByName(name);
            }
            catch (InvalidOperationException ex)
            {
                WriteError(new ErrorRecord(
                    ex,
                    "UnableToAccessProcessByName",
                    ErrorCategory.InvalidOperation,
                    name));
                continue;
            }
        }
    }
}

```

```
        WriteObject(processes, true);
    } // foreach (string name...
} // else
} // ProcessRecord

#endregion Overrides
} // End GetProcCommand class.

#endregion GetProcCommand
}
```

See Also

- [Writing a Windows PowerShell Cmdlet](#)

GetProcessSample05 Sample

Article • 09/15/2023

This sample shows a complete version of the Get-Proc cmdlet.

How to build the sample using Visual Studio.

1. Open Windows Explorer and navigate to the GetProcessSample05 directory under the Samples directory.

With the Windows PowerShell 2.0 SDK installed, navigate to the GetProcessSample05 folder. The default location is `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0\Samples\sysmgmt\WindowsPowerShell\csharp\GetProcessSample05`.

2. Double-click the icon for the solution (.sln) file. This opens the sample project in Visual Studio.
3. In the **Build** menu, select **Build Solution** to build the library for the sample in the default `\bin` or `\bin\debug` folders.

How to run the sample

1. Create the following module folder:

```
[user]\Documents\WindowsPowerShell\Modules\GetProcessSample05
```

2. Copy the sample assembly to the module folder.
3. Start Windows PowerShell.
4. Run the following command to load the assembly into Windows PowerShell:

```
Import-Module getprocesssample05
```

5. Run the following command to run the cmdlet:

```
Get-Proc
```

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Declaring a cmdlet class using the Cmdlet attribute.
- Declaring a cmdlet parameter using the Parameter attribute.
- Specifying positions for parameters.
- Specifying that parameters can take input from the pipeline. The input can be taken from an object or a value from a property of an object whose property name is the same as the parameter name.
- Declaring a validation attribute for the parameter input.
- Handling errors and exceptions.
- Writing debug messages.

Example

This sample shows how to create a cmdlet that displays a list of specified processes.

C#

```
namespace Microsoft.Samples.PowerShell.Commands
{
    using System;
    using System.Collections.Generic;
    using System.Diagnostics;
    using System.Management.Automation;      // Windows PowerShell namespace.
    using System.Security.Permissions;
    using Win32Exception = System.ComponentModel.Win32Exception;
    #region GetProcCommand

        /// <summary>
        /// This class implements the Get-Proc cmdlet.
        /// </summary>
        [Cmdlet(VerbsCommon.Get, "Proc",
            DefaultParameterSetName = "ProcessName")]
    public class GetProcCommand : PSCmdlet
    {
        #region Fields
        /// <summary>
        /// The names of the processes to act on.
        /// </summary>
        private string[] processNames;
```

```

///<summary>
/// The identifiers of the processes to act on.
///</summary>
private int[] processIds;

///<summary>
/// The process objects to act on.
///</summary>
private Process[] inputObjects;

#endregion Fields

#region Parameters

///<summary>
/// Gets or sets the list of process names on
/// which the Get-Proc cmdlet will work.
///</summary>
[Parameter(
    Position = 0,
    ParameterSetName = "ProcessName",
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true)]
[ValidateNotNullOrEmpty]
public string[] Name
{
    get { return this.processNames; }
    set { this.processNames = value; }
}

///<summary>
/// Gets or sets the list of process identifiers on
/// which the Get-Proc cmdlet will work.
///</summary>
[Parameter(
    ParameterSetName = "Id",
    Mandatory = true,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true,
    HelpMessage = "The unique id of the process to get.")]
public int[] Id
{
    get { return this.processIds; }
    set { this.processIds = value; }
}

///<summary>
/// Gets or sets Process objects directly. If the input is a
/// stream of [collection of] Process objects, the cmdlet bypasses the
/// ProcessName and Id parameters and reads the Process objects
/// directly. This allows the cmdlet to deal with processes that have
/// wildcard characters in their name.
///<value>Process objects</value>
///</summary>
[Parameter(

```

```

ParameterSetName = "InputObject",
Mandatory = true,
ValueFromPipeline = true)]
public Process[] Input
{
    get { return this.inputObjects; }
    set { this.inputObjects = value; }
}

#endregion Parameters

#region Cmdlet Overrides

/// <summary>
/// The ProcessRecord method calls the Process.GetProcesses
/// method to retrieve the processes. Then, the WriteObject
/// method writes the associated processes to the pipeline.
/// </summary>
protected override void ProcessRecord()
{
    List<Process> matchingProcesses;

    WriteDebug("Obtaining the list of matching process objects.");

    switch (ParameterSetName)
    {
        case "Id":
            matchingProcesses = this.GetMatchingProcessesById();
            break;
        case "ProcessName":
            matchingProcesses = this.GetMatchingProcessesByName();
            break;
        case "InputObject":
            matchingProcesses = this.GetProcessesByInput();
            break;
        default:
            ThrowTerminatingError(
                new ErrorRecord(
                    new ArgumentException("Bad ParameterSetName"),
                    "UnableToAccessProcessList",
                    ErrorCategory.InvalidOperation,
                    null));
            return;
    } // switch (ParameterSetName)

    WriteDebug("Outputting the matching process objects.");

    matchingProcesses.Sort(ProcessComparison);

    foreach (Process process in matchingProcesses)
    {
        WriteObject(process);
    }
} // ProcessRecord

```

```
#endregion Overrides

#region protected Methods and Data

/// <summary>
/// Retrieves the list of all processes matching the ProcessName
/// parameter and generates a non-terminating error for each
/// specified process name which is not found even though the name
/// contains no wildcards.
/// </summary>
/// <returns>The matching processes.</returns>
[EnvironmentPermissionAttribute(
    SecurityAction.LinkDemand,
    Unrestricted = true)]
private List<Process> GetMatchingProcessesByName()
{
    new EnvironmentPermission(
        PermissionState.Unrestricted).Assert();

    List<Process> allProcesses =
        new List<Process>(Process.GetProcesses());

    // The keys dictionary is used for rapid lookup of
    // processes that are already in the matchingProcesses list.
    Dictionary<int, byte> keys = new Dictionary<int, byte>();

    List<Process> matchingProcesses = new List<Process>();

    if (null == this.processNames)
    {
        matchingProcesses.AddRange(allProcesses);
    }
    else
    {
        foreach (string pattern in this.processNames)
        {
            WriteVerbose("Finding matches for process name \""
                + pattern + "\")");

            // WildCard search on the available processes
            WildcardPattern wildcard =
                new WildcardPattern(
                    pattern,
                    WildcardOptions.IgnoreCase);

            bool found = false;

            foreach (Process process in allProcesses)
            {
                if (!keys.ContainsKey(process.Id))
                {
                    string processName = SafeGetProcessName(process);

                    // Remove the process from the allProcesses list
                    // so that it is not tested again.
                }
            }
        }
    }
}
```

```

        if (processName.Length == 0)
        {
            allProcesses.Remove(process);
        }

        // Perform a wildcard search on this particular
        // process name and check whether it matches the
        // pattern specified.
        if (!wildcard.IsMatch(processName))
        {
            continue;
        }

        WriteDebug("Found matching process id "
            + process.Id + ".");

        // A match is found.
        found = true;

        // Store the process identifier so that the same
process
        // is not added twice.
        keys.Add(process.Id, 0);

        // Add the process to the processes list.
        matchingProcesses.Add(process);
    }
} // foreach (Process...

if (!found &&
    !WildcardPattern.ContainsWildcardCharacters(pattern))
{
    WriteError(new ErrorRecord(
        new ArgumentException("Cannot find process name "
            + "\"" + pattern + "\"."),
        "ProcessNameNotFound",
        ErrorCategory.ObjectNotFound,
        pattern));
}

} // foreach (string...
} // if (null...

return matchingProcesses;
} // GetMatchingProcessesByName

/// <summary>
/// Returns the name of a process. If an error occurs, a blank
/// string is returned.
/// </summary>
/// <param name="process">The process whose name is
/// returned.</param>
/// <returns>The name of the process.</returns>
[EnvironmentPermissionAttribute(
    SecurityAction.LinkDemand, Unrestricted = true)]
protected static string SafeGetProcessName(Process process)

```

```

{
    new EnvironmentPermission(PermissionState.Unrestricted).Assert();
    string name = String.Empty;

    if (process != null)
    {
        try
        {
            return process.ProcessName;
        }
        catch (Win32Exception)
        {
        }
        catch (InvalidOperationException)
        {
        }
    }

    return name;
} // SafeGetProcessName

#endregion Cmdlet Overrides

#region Private Methods

/// <summary>
/// Function to sort by process name first, and then by
/// the process identifier.
/// </summary>
/// <param name="x">First process object.</param>
/// <param name="y">Second process object.</param>
/// <returns>
/// Returns less than zero if x is less than y,
/// greater than 0 if x is greater than y, and 0 if x == y.
/// </returns>
private static int ProcessComparison(Process x, Process y)
{
    int diff = String.Compare(
        SafeGetProcessName(x),
        SafeGetProcessName(y),
        StringComparison.CurrentCultureIgnoreCase);

    if (0 != diff)
    {
        return diff;
    }
    else
    {
        return x.Id.CompareTo(y.Id);
    }
}

/// <summary>
/// Retrieves the list of all processes matching the Id
/// parameter and generates a non-terminating error for

```

```

/// each specified process identifier which is not found.
/// </summary>
/// <returns>
/// An array of processes that match the given identifier.
/// </returns>
[EnvironmentPermissionAttribute(
    SecurityAction.LinkDemand,
    Unrestricted = true)]
private List<Process> GetMatchingProcessesById()
{
    new EnvironmentPermission(
        PermissionState.Unrestricted).Assert();

    List<Process> matchingProcesses = new List<Process>();

    if (null != this.processIds)
    {
        // The keys dictionary is used for rapid lookup of the
        // processes already in the matchingProcesses list.
        Dictionary<int, byte> keys = new Dictionary<int, byte>();

        foreach (int processId in this.processIds)
        {
            WriteVerbose("Finding match for process id "
                + processId + ".");

            if (!keys.ContainsKey(processId))
            {
                Process process;
                try
                {
                    process = Process.GetProcessById(processId);
                }
                catch (ArgumentException ex)
                {
                    WriteError(new ErrorRecord(
                        ex,
                        "ProcessIdNotFound",
                        ErrorCategory.ObjectNotFound,
                        processId));
                    continue;
                }

                WriteDebug("Found matching process.");
                matchingProcesses.Add(process);
                keys.Add(processId, 0);
            }
        }
    }

    return matchingProcesses;
} // GetMatchingProcessesById

```

```

/// Retrieves the list of all processes matching the InputObject
/// parameter.
/// </summary>
/// <returns>The matching processes.</returns>
[EnvironmentPermissionAttribute(
    SecurityAction.LinkDemand,
    Unrestricted = true)]
private List<Process> GetProcessesByInput()
{
    new EnvironmentPermission(
        PermissionState.Unrestricted).Assert();

    List<Process> matchingProcesses = new List<Process>();

    if (null != this.Input)
    {
        // The keys dictionary is used for rapid lookup of the
        // processes already in the matchingProcesses list.
        Dictionary<int, byte> keys = new Dictionary<int, byte>();

        foreach (Process process in this.Input)
        {
            WriteVerbose("Refreshing process object.");

            if (!keys.ContainsKey(process.Id))
            {
                try
                {
                    process.Refresh();
                }
                catch (Win32Exception)
                {
                }
                catch (InvalidOperationException)
                {
                }

                matchingProcesses.Add(process);
            }
        }
    }

    return matchingProcesses;
} // GetProcessesByInput
#endregion Private Methods
} // End GetProcCommand class.

#region GetProcCommand
}

```

See Also

- Writing a Windows PowerShell Cmdlet

StopProcessSample01 Sample

Article • 09/15/2023

This sample shows how to write a cmdlet that requests feedback from the user before it attempts to stop a process, and how to implement a `PassThru` parameter indicating that the user wants the cmdlet to return an object. This cmdlet is similar to the `Stop-Process` cmdlet provided by Windows PowerShell 2.0.

How to build the sample by using Visual Studio

1. With the Windows PowerShell 2.0 SDK installed, navigate to the StopProcessSample01 folder. The default location is `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0\Samples\sysmgmt\WindowsPowerShell\csharp\StopProcessSample01`.
2. Double-click the icon for the solution (.sln) file. This opens the sample project in Microsoft Visual Studio.
3. In the **Build** menu, select **Build Solution** to build the library for the sample in the default `\bin` or `\bin\debug` folders.

How to run the sample

1. Create the following module folder:
`[user]\Documents\WindowsPowerShell\Modules\StopProcessSample01`
2. Copy the sample assembly to the module folder.
3. Start Windows PowerShell.
4. Run the following command to load the assembly into Windows PowerShell:
`Import-Module stopprocesssample01`
5. Run the following command to run the cmdlet:
`Stop-Proc`

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Declaring a cmdlet class by using the `Cmdlet` attribute.
- Declaring a cmdlet parameters by using the `Parameter` attribute.
- Calling the `ShouldProcess` method to request confirmation.
- Implementing a `PassThru` parameter that indicates if the user wants the cmdlet to return an object. By default, this cmdlet does not return an object to the pipeline.

Example

This sample shows how to implement a `PassThru` parameter that indicates that the user wants the cmdlet to return an object, and how to request user feedback by calls to the `ShouldProcess` and `ShouldContinue` methods.

C#

```
using System;
using System.Diagnostics;
using System.Collections;
using Win32Exception = System.ComponentModel.Win32Exception;
using System.Management.Automation;    // Windows PowerShell namespace
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Commands
{
    #region StopProcCommand

        /// <summary>
        /// This class implements the Stop-Proc cmdlet.
        /// </summary>
        [Cmdlet(VerbsLifecycle.Stop, "Proc",
            SupportsShouldProcess = true)]
        public class StopProcCommand : Cmdlet
    {
        #region Parameters

            /// <summary>
            /// This parameter provides the list of process names on
            /// which the Stop-Proc cmdlet will work.
            /// </summary>
            [Parameter(
```

```

        Position = 0,
        Mandatory = true,
        ValueFromPipeline = true,
        ValueFromPipelineByPropertyName = true
    )]
public string[] Name
{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;

/// <summary>
/// This parameter overrides the ShouldContinue call to force
/// the cmdlet to stop its operation. This parameter should always
/// be used with caution.
/// </summary>
[Parameter]
public SwitchParameter Force
{
    get { return force; }
    set { force = value; }
}
private bool force;

/// <summary>
/// This parameter indicates that the cmdlet should return
/// an object to the pipeline after the processing has been
/// completed.
/// </summary>
[Parameter]
public SwitchParameter PassThru
{
    get { return passThru; }
    set { passThru = value; }
}
private bool passThru;

#endregion Parameters

#region Cmdlet Overrides

/// <summary>
/// The ProcessRecord method does the following for each of the
/// requested process names:
/// 1) Check that the process is not a critical process.
/// 2) Attempt to stop that process.
/// If no process is requested then nothing occurs.
/// </summary>
protected override void ProcessRecord()
{
    foreach (string name in processNames)
    {
        // For every process name passed to the cmdlet, get the
associated

```

```

// processes.
// Write a non-terminating error for failure to retrieve
// a process.
Process[] processes;

try
{
    processes = Process.GetProcessesByName(name);
}
catch (InvalidOperationException ioe)
{
    WriteError(new
ErrorRecord(ioe, "UnableToAccessProcessByName",
            ErrorCategory.InvalidOperation, name));

    continue;
}

// Try to stop the processes that have been retrieved.
foreach (Process process in processes)
{
    string processName;

    try
    {
        processName = process.ProcessName;
    }
    catch (Win32Exception e)
    {
        WriteError(new ErrorRecord(e, "ProcessNameNotFound",
                                ErrorCategory.ReadError,
process));
        continue;
    }

    // Confirm the operation with the user first.
    // This is always false if the WhatIf parameter is set.
    if
(!ShouldProcess(string.Format(CultureInfo.CurrentCulture, "{0} ({1})",
processName,
                    process.Id)))
    {
        continue;
    }

    // Make sure that the user really wants to stop a
critical
    // process that could possibly stop the computer.
    bool criticalProcess =

criticalProcessNames.Contains(processName.ToLower(CultureInfo.CurrentCulture
)));
}

if (criticalProcess &&!force)
{

```

```

        string message = String.Format
            (CultureInfo.CurrentCulture,
                "The process \"\{0}\\" is a critical process
and should not be stopped. Are you sure you wish to stop the process?", 
                processName);

        // It is possible that the ProcessRecord method is
called
        // multiple times when objects are received as inputs
from
        // the pipeline. So to retain YesToAll and NoToAll
input that
        // the user may enter across multiple calls to this
function,
        // they are stored as private members of the cmdlet.
if (!ShouldContinue(message, "Warning!",
                    ref yesToAll, ref noToAll))
{
    continue;
}
} // if (criticalProcess...

// Stop the named process.
try
{
    process.Kill();
}
catch (Exception e)
{
    if ((e is Win32Exception) || (e is SystemException)
||

        (e is InvalidOperationException))
    {
        // This process could not be stopped so write
        // a non-terminating error.
        WriteError(new ErrorRecord(e,
"CouldNotStopProcess",
                                ErrorCategory.CloseError,
process));
        continue;
    } // if ((e is...
    else throw;
} // catch

// If the PassThru parameter is
// specified, return the terminated process.
if (passThru)
{
    WriteObject(process);
}
} // foreach (Process...
} // foreach (string...
} // ProcessRecord

#endregion Cmdlet Overrides

```

```
#region Private Data

private bool yesToAll, noToAll;

/// <summary>
/// Partial list of critical processes that should not be
/// stopped. Lower case is used for case insensitive matching.
/// </summary>
private ArrayList criticalProcessNames = new ArrayList(
    new string[] { "system", "winlogon", "spoolsv" }
);

#endregion Private Data

} // StopProcCommand

#endregion StopProcCommand
}
```

See Also

- [Writing a Windows PowerShell Cmdlet](#)

StopProcessSample02 Sample

Article • 09/15/2023

This sample shows how to write a cmdlet that writes debug (WriteDebug), verbose (WriteVerbose), and warning (WriteWarning) messages while stopping processes on the local computer. This cmdlet is similar to the `Stop-Process` cmdlet provided by Windows PowerShell 2.0.

How to build the sample by using Visual Studio

1. Open Windows Internet Explorer and navigate to the StopProcessSample02 directory under the Samples directory.

With the Windows PowerShell 2.0 SDK installed, navigate to the StopProcessSample02 folder. The default location is `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0\Samples\sysmgmt\WindowsPowerShell\csharp\StopProcessSample02`.

2. Double-click the icon for the solution (.sln) file. This opens the sample project in Microsoft Visual Studio.
3. In the **Build** menu, select **Build Solution** to build the library for the sample in the default `\bin` or `\bin\debug` folders.

How to run the sample

1. Create the following module folder:

```
[user]\Documents\WindowsPowerShell\Modules\StopProcessSample02
```

2. Copy the sample assembly to the module folder.
3. Start Windows PowerShell.
4. Run the following command to load the assembly into Windows PowerShell:

```
Import-Module stopprocesssample02
```

5. Run the following command to run the cmdlet:

```
Stop-Proc
```

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Declaring a cmdlet class by using the `Cmdlet` attribute.
- Declaring a cmdlet parameters by using the `Parameter` attribute.
- Writing verbose messages. For more information about the method used to write verbose messages, see [System.Management.Automation.Cmdlet.WriteVerbose](#).
- Writing error messages. For more information about the method used to write error messages, see [System.Management.Automation.Cmdlet.WriteError](#).
- Writing warning messages. For more information about the method used to write warning messages, see [System.Management.Automation.Cmdlet.WriteWarning](#).

Example

This sample shows how to write debug, verbose, and warning messages by using the `WriteDebug`, `WriteVerbose`, and `WriteWarning` methods.

C#

```
using System;
using System.Diagnostics;
using System.Collections;
using Win32Exception = System.ComponentModel.Win32Exception;
using System.Management.Automation; //Windows PowerShell
namespace
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Commands
{
    #region StopProcCommand

        /// <summary>
        /// This class implements the Stop-Proc cmdlet.
        /// </summary>
        [Cmdlet(VerbsLifecycle.Stop, "Proc",
            SupportsShouldProcess = true)]
        public class StopProcCommand : Cmdlet
    {
```

```

#region Parameters

/// <summary>
/// This parameter provides the list of process names on
/// which the Stop-Proc cmdlet will work.
/// </summary>
[Parameter(
    Position = 0,
    Mandatory = true,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true
)]
public string[] Name
{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;

/// <summary>
/// This parameter overrides the ShouldContinue call to force
/// the cmdlet to stop its operation. This parameter should always
/// be used with caution.
/// </summary>
[Parameter]
public SwitchParameter Force
{
    get { return force; }
    set { force = value; }
}
private bool force;

/// <summary>
/// This parameter indicates that the cmdlet should return
/// an object to the pipeline after the processing has been
/// completed.
/// </summary>
[Parameter]
public SwitchParameter PassThru
{
    get { return passThru; }
    set { passThru = value; }
}
private bool passThru;

#endregion Parameters

#region Cmdlet Overrides

/// <summary>
/// The ProcessRecord method does the following for each of the
/// requested process names:
/// 1) Check that the process is not a critical process.
/// 2) Attempt to stop that process.
/// If no process is requested then nothing occurs.

```

```
/// </summary>
protected override void ProcessRecord()
{
    foreach (string name in processNames)
    {
        string message = null;

        // For every process name passed to the cmdlet, get the
associated
        // processes.
        // Write a non-terminating error for failure to retrieve
        // a process.

        // Write a user-friendly verbose message to the pipeline.
These
information
will
        // messages are intended to give the user detailed
        // on the operations performed by the cmdlet. These messages
        // appear with the -Verbose option.
        message = String.Format(CultureInfo.CurrentCulture,
                               "Attempting to stop process \"\{0\}\".", name);
        WriteVerbose(message);

        Process[] processes;

        try
        {
            processes = Process.GetProcessesByName(name);
        }
        catch (InvalidOperationException ioe)
        {
            WriteError(new ErrorRecord(ioe,
                                      "UnableToAccessProcessByName",
                                      ErrorCategory.InvalidOperation,
                                      name));
            continue;
        }

        // Try to stop the processes that have been retrieved.
        foreach (Process process in processes)
        {
            string processName;

            try
            {
                processName = process.ProcessName;
            }
            catch (Win32Exception e)
            {
                WriteError(new ErrorRecord(e, "ProcessNameNotFound",
                                          ErrorCategory.ObjectNotFound,
process));
                continue;
            }
        }
    }
}
```

```

        // Write a debug message to the host that can be used
when
        // troubleshooting a problem. All debug messages will
appear
        // with the -Debug option.
message = String.Format(CultureInfo.CurrentCulture,
                        "Acquired name for pid {0} : \"{1}\",
process.Id, processName);
WriteDebug(message);

        // Confirm the operation first.
        // This is always false if the WhatIf parameter is
specified.
        if
(!ShouldProcess(string.Format(CultureInfo.CurrentCulture,
                            "{0} ({1})",
                            processName, process.Id)))
{
    continue;
}

        // Make sure that the user really wants to stop a
critical
        // process that can possibly stop the computer.
        bool criticalProcess =
criticalProcessNames.Contains(processName.ToLower(CultureInfo.CurrentCulture
));

        if (criticalProcess && !force)
{
    message = String.Format(CultureInfo.CurrentCulture,
                            "The process \"{0}\" is a critical
process and should not be stopped. Are you sure you wish to stop the
process?",

processName);

        // It is possible that the ProcessRecord method is
called
        // multiple times when objects are received as inputs
from
        // the pipeline. So to retain YesToAll and NoToAll
input that
        // the user may enter across multiple calls to this
function,
        // they are stored as private members of the cmdlet.
        if (!ShouldContinue(message, "Warning!",
                            ref yesToAll, ref noToAll))
{
    continue;
}
} // if (criticalProcess...

        // Display a warning message if the cmdlet is stopping a
// critical process.

```

```

    if (criticalProcess)
    {
        message = String.Format(CultureInfo.CurrentCulture,
                               "Stopping the critical process \""
{0}\",",
                               processName);
        WriteWarning(message);
    } // if (criticalProcess...

    // Stop the named process.
    try
    {
        process.Kill();
    }
    catch (Exception e)
    {
        if ((e is Win32Exception) || (e is SystemException)
||

(e is InvalidOperationException))
        {
            // This process could not be stopped so write
            // a non-terminating error.
            WriteError(new ErrorRecord(
                e,
                "CouldNotStopProcess",
                ErrorCategory.CloseError,
                process)
            );
            continue;
        } // if ((e is...
        else throw;
    } // catch

    message = String.Format(CultureInfo.CurrentCulture,
                           "Stopped process \"{0}\", pid {1}.",
                           processName, process.Id);

    WriteVerbose(message);

    // If the PassThru parameter is specified,
    // return the terminated process object to the pipeline.
    if (passThru)
    {
        message = String.Format(CultureInfo.CurrentCulture,
                               "Writing process \"{0}\" to pipeline",
                               processName);
        WriteDebug(message);
        WriteObject(process);
    } // if (passThru...
} // foreach (Process...
} // foreach (string...
} // ProcessRecord

#endregion Cmdlet Overrides

```

```
#region Private Data

private bool yesToAll, noToAll;

/// <summary>
/// Partial list of critical processes that should not be
/// stopped. Lower case is used for case insensitive matching.
/// </summary>
private ArrayList criticalProcessNames = new ArrayList(
    new string[] { "system", "winlogon", "spoolsv" }
);

#endregion Private Data

} // StopProcCommand

#endregion StopProcCommand
}
```

See Also

- [Writing a Windows PowerShell Cmdlet](#)

StopProcessSample03 Sample

Article • 09/15/2023

This sample shows how to write a cmdlet whose parameters have aliases and whose parameters support wildcard characters. This cmdlet is similar to the `Stop-Process` cmdlet provided by Windows PowerShell 2.0.

How to build the sample by using Visual Studio

1. With the Windows PowerShell 2.0 SDK installed, navigate to the StopProcessSample03 folder. The default location is `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0\Samples\sysmgmt\WindowsPowerShell\csharp\StopProcessSample03`.
2. Double-click the icon for the solution (.sln) file. This opens the sample project in Microsoft Visual Studio.
3. In the **Build** menu, select **Build Solution** to build the library for the sample in the default `\bin` or `\bin\debug` folders.

How to run the sample

1. Create the following module folder:
`[user]\Documents\WindowsPowerShell\Modules\StopProcessSample03`
2. Copy the sample assembly to the module folder.
3. Start Windows PowerShell.
4. Run the following command to load the assembly into Windows PowerShell:

```
Import-Module stopprocesssample03
```

5. Run the following command to run the cmdlet:

```
Stop-Proc
```

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Declaring a cmdlet class by using the Cmdlet attribute.
- Declaring a cmdlet parameters by using the Parameter attribute.
- Adding aliases to parameter declarations..
- Adding wildcard support to parameters.

Example

This sample shows how to declare parameter aliases and support wildcards.

C#

```
using System;
using System.Diagnostics;
using System.Collections;
using Win32Exception = System.ComponentModel.Win32Exception;
using System.Management.Automation; //Windows PowerShell
namespace
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Commands
{

    #region StopProcCommand

        /// <summary>
        /// This class implements the Stop-Proc cmdlet.
        /// </summary>
        [Cmdlet(VerbsLifecycle.Stop, "Proc",
            SupportsShouldProcess = true)]
        public class StopProcCommand : Cmdlet
    {
        #region Parameters

            /// <summary>
            /// This parameter provides the list of process names on
            /// which the Stop-Proc cmdlet will work.
            /// </summary>
            [Parameter(
                Position = 0,
                Mandatory = true,
                ValueFromPipeline = true,
```

```

        ValueFromPipelineByPropertyName = true,
        HelpMessage = "The name of one or more processes to stop.
Wildcards are permitted."
    )]
[Alias("ProcessName")]
public string[] Name
{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;

/// <summary>
/// This parameter overrides the ShouldContinue call to force
/// the cmdlet to stop its operation. This parameter should always
/// be used with caution.
/// </summary>
[Parameter]
public SwitchParameter Force
{
    get { return force; }
    set { force = value; }
}
private bool force;

/// <summary>
/// This parameter indicates that the cmdlet should return
/// an object to the pipeline after the processing has been
/// completed.
/// </summary>
[Parameter(
    ValueFromPipelineByPropertyName = true,
    HelpMessage = "If set, the process(es) will be passed to the
pipeline after stopped."]
)
public SwitchParameter PassThru
{
    get { return passThru; }
    set { passThru = value; }
}
private bool passThru;

#endregion Parameters

#region Cmdlet Overrides
/// <summary>
/// The ProcessRecord method does the following for each of the
/// requested process names:
/// 1) Check that the process is not a critical process.
/// 2) Attempt to stop that process.
/// If no process is requested then nothing occurs.
/// </summary>
protected override void ProcessRecord()
{
    Process[] processes = null;
}

```

```
try
{
    processes = Process.GetProcesses();
}
catch (InvalidOperationException ioe)
{
    base.ThrowTerminatingError(new ErrorRecord(ioe,
        "UnableToAccessProcessList",
        ErrorCategory.InvalidOperation,
        null));
}

// For every process name passed to the cmdlet, get the
associated
// processes.
// Write a non-terminating error for failure to retrieve
// a process.
foreach (string name in processNames)
{
    // Write a user-friendly verbose message to the pipeline.
These
information
will
    // messages are intended to give the user detailed
    // on the operations performed by the cmdlet. These messages
    // appear with the -Verbose option.
    string message = String.Format(CultureInfo.CurrentCulture,
        "Attempting to stop process \'{0}\'.",
name);
    WriteVerbose(message);

    // Validate the process name against a wildcard pattern.
    // If the name does not contain any wildcard patterns, it
    // will be treated as an exact match.
    WildcardOptions options = WildcardOptions.IgnoreCase |
                                WildcardOptions.Compiled;
    WildcardPattern wildcard = new WildcardPattern(name,options);

    foreach (Process process in processes)
    {
        string processName;

        try
        {
            processName = process.ProcessName;
        }
        catch (Win32Exception e)
        {
            WriteError(new ErrorRecord(
                e, "ProcessNameNotFound",
                ErrorCategory.ObjectNotFound,
                process));
        };
    };
}
```

```

        continue;
    }

    // Write a debug message to the host that can be used
when
    // troubleshooting a problem. All debug messages will
appear
    // with the -Debug option.
    message = String.Format(CultureInfo.CurrentCulture,
        "Acquired name for pid {0} : \"{1}\"",
        process.Id, processName);
    WriteDebug(message);

    // Check to see if this process matches the current
process
    // name pattern. Skip this process if it does not.
    if (!wildcard.IsMatch(processName))
    {
        continue;
    }

    // Stop the process.
    SafeStopProcess(process);
} // foreach (Process...
} // foreach (string...
} // ProcessRecord

#endregion Cmdlet Overrides

#region Helper Methods

/// <summary>
/// Safely stops a named process. Used as standalone function
/// to declutter the ProcessRecord method.
/// </summary>
/// <param name="process">The process to stop.</param>
private void SafeStopProcess(Process process)
{
    string processName = null;
    try
    {
        processName = process.ProcessName;
    }
    catch (Win32Exception e)
    {
        WriteError(new ErrorRecord(e, "ProcessNameNotFound",
            ErrorCategory.ObjectNotFound, process));
        return;
    }

    string message = null;

    // Confirm the operation first.
    // This is always false if the WhatIf parameter is specified.
    if (!ShouldProcess(string.Format(CultureInfo.CurrentCulture,

```



```

        process)
    );
    return;
} // if ((e is...
else throw;
} // catch

message = String.Format(CultureInfo.CurrentCulture,
    "Stopped process \"{0}\", pid {1}.",
    processName, process.Id);

WriteVerbose(message);

// If the PassThru parameter is specified,
// return the terminated process to the pipeline.
if (passThru)
{
    message = String.Format(CultureInfo.CurrentCulture,
        "Writing process \"{0}\" to pipeline",
        processName);
    WriteDebug(message);
    WriteObject(process);
} // if (passThru...
} // SafeStopProcess

#endregion Helper Methods

#region Private Data

private bool yesToAll, noToAll;

/// <summary>
/// Partial list of the critical processes that should not be
/// stopped. Lower case is used for case insensitive matching.
/// </summary>
private ArrayList criticalProcessNames = new ArrayList(
    new string[] { "system", "winlogon", "spoolsv" })
;

#endregion Private Data

} // StopProcCommand

#endregion StopProcCommand
} // namespace Microsoft.Samples.PowerShell.Commands

```

See Also

- [Writing a Windows PowerShell Cmdlet](#)

StopProcessSample04 Sample

Article • 09/15/2023

This sample shows how to write a cmdlet that declares parameter sets, specifies the default parameter set, and can accept an input object. This cmdlet is similar to the `Stop-Process` cmdlet provided by Windows PowerShell 2.0.

How to build the sample by using Visual Studio

1. With the Windows PowerShell 2.0 SDK installed, navigate to the StopProcessSample04 folder. The default location is `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0\Samples\sysmgmt\WindowsPowerShell\csharp\StopProcessSample04`.
2. Double-click the icon for the solution (.sln) file. This opens the sample project in Microsoft Visual Studio.
3. In the **Build** menu, select **Build Solution** to build the library for the sample in the default `\bin` or `\bin\debug` folders.

How to run the sample

1. Create the following module folder:
`[user]\Documents\WindowsPowerShell\Modules\StopProcessSample04`
2. Copy the sample assembly to the module folder.
3. Start Windows PowerShell.
4. Run the following command to load the assembly into Windows PowerShell:

```
Import-Module stopprocesssample04
```

5. Run the following command to run the cmdlet:

```
Stop-Proc
```

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Declaring a cmdlet class by using the Cmdlet attribute.
- Declaring a cmdlet parameters by using the Parameter attribute.
- Adding a parameter that accepts input object.
- Adding parameters to parameter sets
- Specifying the default parameter set.

Example

The following code shows an implementation of the Stop-Proc cmdlet that declare parameter sets, specifies the default parameter set, and can accept an input object.

This sample shows the input object, how to declare parameter sets, and how to specify the default parameter set to use.

C#

```
using System;
using System.Diagnostics;
using System.Collections;
using Win32Exception = System.ComponentModel.Win32Exception;
using System.Management.Automation; //Windows PowerShell
namespace
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Commands
{
    #region StopProcCommand

    /// <summary>
    /// This class implements the Stop-Proc cmdlet.
    /// </summary>
    [Cmdlet(VerbsLifecycle.Stop, "Proc",
        DefaultParameterSetName = "ProcessId",
        SupportsShouldProcess = true)]
    public class StopProcCommand : PSCmdlet
    {
        #region Parameters
```

```

/// <summary>
/// This parameter provides the list of process names on
/// which the Stop-Proc cmdlet will work.
/// </summary>
[Parameter(
    Position = 0,
    ParameterSetName = "ProcessName",
    Mandatory = true,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true,
    HelpMessage = "The name of one or more processes to stop.
Wildcards are permitted."
)]
[Alias("ProcessName")]
public string[] Name
{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;

/// <summary>
/// This parameter overrides the ShouldContinue call to force
/// the cmdlet to stop its operation. This parameter should always
/// be used with caution.
/// </summary>
[Parameter]
public SwitchParameter Force
{
    get { return force; }
    set { force = value; }
}
private bool force;

/// <summary>
/// This parameter indicates that the cmdlet should return
/// an object to the pipeline after the processing has been
/// completed.
/// </summary>
[Parameter(
    HelpMessage = "If set the process(es) will be passed to the
pipeline after stopped."
)]
public SwitchParameter PassThru
{
    get { return passThru; }
    set { passThru = value; }
}
private bool passThru;

/// This parameter provides the list of process identifiers on
/// which the Stop-Proc cmdlet will work.
[Parameter(
    ParameterSetName = "ProcessId",
    Mandatory = true,

```

```

        ValueFromPipelineByPropertyName = true,
        ValueFromPipeline = true
    )]
[Alias("ProcessId")]
public int[] Id
{
    get { return processIds; }
    set { processIds = value; }
}
private int[] processIds;

/// <summary>
/// This parameter accepts an array of Process objects from the
/// the pipeline. This object contains the processes to stop.
/// </summary>
/// <value>Process objects</value>
[Parameter(
    ParameterSetName = "InputObject",
    Mandatory = true,
    ValueFromPipeline = true)]
public Process[] InputObject
{
    get { return inputObject; }
    set { inputObject = value; }
}
private Process[] inputObject;

#endregion Parameters

#region CmdletOverrides

/// <summary>
/// The ProcessRecord method does the following for each of the
/// requested process names:
/// 1) Check that the process is not a critical process.
/// 2) Attempt to stop that process.
/// If no process is requested then nothing occurs.
/// </summary>
protected override void ProcessRecord()
{
    switch (ParameterSetName)
    {
        case "ProcessName":
            ProcessByName();
            break;

        case "ProcessId":
            ProcessById();
            break;

        case "InputObject":
            foreach (Process process in inputObject)
            {
                SafeStopProcess(process);
            }
    }
}

```

```

        break;

    default:
        throw new ArgumentException("Bad ParameterSet Name");
    } // switch (ParameterSetName...
} // ProcessRecord

#endregion Cmdlet Overrides

#region Helper Methods

/// <summary>
/// Returns all processes with matching names.
/// </summary>
/// <param name="processName">
/// The name of the processes to return.
/// </param>
/// <param name="allProcesses">An array of all
/// computer processes.</param>
/// <returns>An array of matching processes.</returns>
internal ArrayList SafeGetProcessesByName(string processName,
                                         ref ArrayList allProcesses)
{
    // Create and array to store the matching processes.
    ArrayList matchingProcesses = new ArrayList();

    // Create the wildcard for pattern matching.
    WildcardOptions options = WildcardOptions.IgnoreCase |
                               WildcardOptions.Compiled;
    WildcardPattern wildcard = new WildcardPattern(processName,
options);

    // Walk all of the machine processes.
    foreach(Process process in allProcesses)
    {
        string processNameToMatch = null;
        try
        {
            processNameToMatch = process.ProcessName;
        }
        catch (Win32Exception e)
        {
            // Remove the process from the list so that it is not
            // checked again.
            allProcesses.Remove(process);

            string message =
                String.Format(CultureInfo.CurrentCulture, "The
process \'{0}\' could not be found",
                           processName);
            WriteVerbose(message);
            WriteError(new ErrorRecord(e, "ProcessNotFound",
                                      ErrorCategory.ObjectNotFound,
processName));
        }
    }
}

```

```

        continue;
    }

    if (!wildcard.IsMatch(processNameToMatch))
    {
        continue;
    }

    matchingProcesses.Add(process);
} // foreach(Process...

return matchingProcesses;
} // SafeGetProcessesByName

/// <summary>
/// Safely stops a named process. Used as standalone function
/// to declutter the ProcessRecord method.
/// </summary>
/// <param name="process">The process to stop.</param>
private void SafeStopProcess(Process process)
{
    string processName = null;

    try
    {
        processName = process.ProcessName;
    }
    catch (Win32Exception e)
    {
        WriteError(new ErrorRecord(e, "ProcessNotFound",
            ErrorCategory.OpenError, processName));

        return;
    }

    // Confirm the operation first.
    // This is always false if the WhatIf parameter is specified.
    if (!ShouldProcess(string.Format(CultureInfo.CurrentCulture,
        "{0} ({1})", processName, process.Id)))
    {
        return;
    }

    // Make sure that the user really wants to stop a critical
    // process that can possibly stop the computer.
    bool criticalProcess =
criticalProcessNames.Contains(processName.ToLower(CultureInfo.CurrentCulture
));

    string message = null;
    if (criticalProcess && !force)
    {
        message = String.Format(CultureInfo.CurrentCulture,
            "The process \'{0}\' is a
critical process and should not be stopped. Are you sure you wish to stop

```

```

the process?",  

                                processName);  

// It is possible that the ProcessRecord method is called  

// multiple times when objects are received as inputs from  

// the pipeline. So to retain YesToAll and NoToAll input that  

// the user may enter across multiple calls to this function,  

// they are stored as private members of the cmdlet.  

if (!ShouldContinue(message, "Warning!",  

                        ref yesToAll, ref noToAll))  

{  

    return;  

}  

} // if (criticalProcess...  

// Display a warning message if stopping a critical  

// process.  

if (criticalProcess)  

{  

    message =  

        String.Format(CultureInfo.CurrentCulture,  

                    "Stopping the critical process \'{0}\'.",  

                    processName);  

    WriteWarning(message);  

} // if (criticalProcess...  

try  

{  

    // Stop the process.  

    process.Kill();  

}  

catch (Exception e)  

{  

    if ((e is Win32Exception) || (e is SystemException) ||  

        (e is InvalidOperationException))  

    {  

        // This process could not be stopped so write  

        // a non-terminating error.  

        WriteError(new ErrorRecord(e, "CouldNotStopProcess",  

                                    ErrorCategory.CloseError,  

                                    process)  

    );  

        return;  

} // if ((e is...  

else throw;  

} // catch  

// Write a user-level verbose message to the pipeline. These are  

// intended to give the user detailed information on the  

// operations performed by the cmdlet. These messages will  

// appear with the -Verbose option.  

message = String.Format(CultureInfo.CurrentCulture,  

                    "Stopped process \'{0}\', pid {1}.",  

                    processName, process.Id);

```

```

        WriteVerbose(message);

        // If the PassThru parameter is specified, return the terminated
        // process to the pipeline.
        if (passThru)
        {
            // Write a debug message to the host that can be used
            // when troubleshooting a problem. All debug messages
            // will appear with the -Debug option
            message =
                String.Format(CultureInfo.CurrentCulture,
                    "Writing process \"{0}\" to pipeline",
                    processName);
            WriteDebug(message);
            WriteObject(process);
        } // if (passThru..
    } // SafeStopProcess

    /// <summary>
    /// Stop processes based on their names (using the
    /// ParameterSetName as ProcessName)
    /// </summary>
    private void ProcessByName()
    {
        ArrayList allProcesses = null;

        // Get a list of all processes.
        try
        {
            allProcesses = new ArrayList(Process.GetProcesses());
        }
        catch (InvalidOperationException ioe)
        {
            base.ThrowTerminatingError(new ErrorRecord(
                ioe, "UnableToAccessProcessList",
                ErrorCategory.InvalidOperation, null));
        }

        // If a process name is passed to the cmdlet, get
        // the associated processes.
        // Write a non-terminating error for failure to
        // retrieve a process.
        foreach (string name in processNames)
        {
            // The allProcesses array list is passed as a reference
because
            // any process whose name cannot be obtained will be removed
            // from the list so that its not compared the next time.
            ArrayList processes =
                SafeGetProcessesByName(name, ref allProcesses);

            // If no processes were found write a non-
            // terminating error.
            if (processes.Count == 0)
            {

```

```

        WriteError(new ErrorRecord(
            new Exception("Process not found."),
            "ProcessNotFound",
            ErrorCategory.ObjectNotFound,
            name));
    } // if (processes...
    // Otherwise terminate all processes in the list.
    else
    {
        foreach (Process process in processes)
        {
            SafeStopProcess(process);
        } // foreach (Process...
    } // else
} // foreach (string...
} // ProcessByName

/// <summary>
/// Stop processes based on their identifiers (using the
/// ParameterSetName as ProcessIds)
/// </summary>
internal void ProcessById()
{
    foreach (int processId in processIds)
    {
        Process process = null;
        try
        {
            process = Process.GetProcessById(processId);

            // Write a debug message to the host that can be used
            // when troubleshooting a problem. All debug messages
            // will appear with the -Debug option
            string message =
                String.Format(CultureInfo.CurrentCulture,
                    "Acquired process for pid : {0}",
                    process.Id);
            WriteDebug(message);
        }
        catch (ArgumentException ae)
        {
            string
                message = String.Format(CultureInfo.CurrentCulture,
                    "The process id {0} could not be
found",
                processId);
            WriteVerbose(message);
            WriteError(new ErrorRecord(ae, "ProcessIdNotFound",
                ErrorCategory.ObjectNotFound,
                processId));
            continue;
        }

        SafeStopProcess(process);
    } // foreach (int...
}

```

```
    } // ProcessById

    #endregion Helper Methods

    #region Private Data

    private bool yesToAll, noToAll;

    /// <summary>
    /// Partial list of critical processes that should not be
    /// stopped. Lower case is used for case insensitive matching.
    /// </summary>
    private ArrayList criticalProcessNames = new ArrayList(
        new string[] { "system", "winlogon", "spoolsv", "calc" })
    ;

    #endregion Private Data

} // StopProcCommand

#endregion StopProcCommand
}
```

See Also

- [Writing a Windows PowerShell Cmdlet](#)

Events01 Sample

Article • 09/15/2023

This sample shows how to create a cmdlet that allows the user to register for events that are raised by [System.IO.FileSystemWatcher](#). With this cmdlet, users can register an action to execute when a file is created under a specific directory. This sample derives from the [Microsoft.PowerShell.Commands.ObjectEventRegistrationBase](#) base class.

How to build the sample by using Visual Studio

1. With the Windows PowerShell 2.0 SDK installed, navigate to the Events01 folder.
The default location is `C:\Program Files (x86)\Microsoft
SDKs\Windows\v7.0\Samples\sysmgmt\WindowsPowerShell\csharp\Events01`.
2. Double-click the icon for the solution (.sln) file. This opens the sample project in Microsoft Visual Studio.
3. In the **Build** menu, select **Build Solution** to build the library for the sample in the default `\bin` or `\bin\debug` folders.

How to run the sample

1. Create the following module folder:
`[user]\Documents\WindowsPowerShell\Modules\events01`
2. Copy the library file for the sample to the module folder.
3. Start Windows PowerShell.
4. Run the following command to load the cmdlet into Windows PowerShell:

```
PowerShell  
  
Import-Module events01
```

5. Use the Register-FileSystemEvent cmdlet to register an action that will write a message when a file is created under the TEMP directory.

```
PowerShell
```

```
Register-FileSystemEvent $Env:TEMP Created -Filter "*.txt" -Action {  
    Write-Host "A file was created in the TEMP directory" }
```

6. Create a file under the TEMP directory and note that the action is executed (the message is displayed).

This is a sample output that results by following these steps.

Output				
Id	Name	State	HasMoreData	Location
Command				
--	-----	-----	-----	-----

1	26932870-d3b...	NotStarted	False	
Write-Host "A f...				

PowerShell	
<pre>Set-Content \$Env:TEMP\test.txt "This is a test file"</pre>	
Output	
A file was created in the TEMP directory	

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

How to write a cmdlet for event registration

The cmdlet derives from the

[Microsoft.PowerShell.Commands.ObjectEventRegistrationBase](#) class, which provides support for parameters common to the `Register-*Event` cmdlets. Cmdlets that are derived from [Microsoft.PowerShell.Commands.ObjectEventRegistrationBase](#) need only

to define their particular parameters and override the `GetSourceObject` and `GetSourceObjectEventName` abstract methods.

Example

This sample shows how to register for events raised by `System.IO.FileSystemWatcher`.

C#

```
namespace Sample
{
    using System;
    using System.IO;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using Microsoft.PowerShell.Commands;

    [Cmdlet(VerbsLifecycle.Register, "FileSystemEvent")]
    public class RegisterObjectEventCommand : ObjectEventRegistrationBase
    {
        /// <summary>The FileSystemWatcher that exposes the events.
        </summary>
        private FileSystemWatcher fileSystemWatcher = new
        FileSystemWatcher();

        /// <summary>Name of the event to which the cmdlet registers.
        </summary>
        private string eventName = null;

        /// <summary>
        /// Gets or sets the path that will be monitored by the
        FileSystemWatcher.
        /// </summary>
        [Parameter(Mandatory = true, Position = 0)]
        public string Path
        {
            get
            {
                return this.fileSystemWatcher.Path;
            }

            set
            {
                this.fileSystemWatcher.Path = value;
            }
        }

        /// <summary>
        /// Gets or sets the name of the event to which the cmdlet
        registers.
        /// <para>
        /// Currently System.IO.FileSystemWatcher exposes 6 events: Changed,
    }
```

```
Created,
    /// Deleted, Disposed, Error, and Renamed. Check the documentation
of
    /// FileSystemWatcher for details on each event.
    /// </para>
    /// </summary>
[Parameter(Mandatory = true, Position = 1)]
public string EventName
{
    get
    {
        return this.eventName;
    }

    set
    {
        this.eventName = value;
    }
}

/// <summary>
/// Gets or sets the filter that will be user by the
FileSystemWatcher.
/// </summary>
[Parameter(Mandatory = false)]
public string Filter
{
    get
    {
        return this.FileSystemWatcher.Filter;
    }

    set
    {
        this.FileSystemWatcher.Filter = value;
    }
}

/// <summary>
/// Derived classes must implement this method to return the object
that generates
/// the events to be monitored.
/// </summary>
/// <returns> This sample returns an instance of
System.IO.FileSystemWatcher</returns>
protected override object GetSourceObject()
{
    return this.FileSystemWatcher;
}

/// <summary>
/// Derived classes must implement this method to return the name of
the event to
/// be monitored. This event must be exposed by the input object.
/// </summary>
```

```
/// <returns> This sample returns the event specified by the user  
with the -EventName parameter.</returns>  
protected override string GetSourceObjectEventName()  
{  
    return this.eventName;  
}  
}  
}
```

See Also

- [Writing a Windows PowerShell Cmdlet](#)

Writing a Windows PowerShell Module

Article • 09/17/2021

This document is written for administrators, script developers, and cmdlet developers who need to package and distribute their Windows PowerShell cmdlets. By using Windows PowerShell modules, you can package and distribute your Windows PowerShell solutions without using a compiled language.

Windows PowerShell modules enable you to partition, organize, and abstract your Windows PowerShell code into self-contained, reusable units. With these reusable units, you can easily share your modules directly with others. If you are a script developer, you can also repackage third-party modules to create custom script-based applications. Modules, similar to modules in other scripting languages such as Perl and Python, enable production-ready scripting solutions that use reusable, redistributable components, with the added benefit of enabling you to repackage and abstract multiple components to create custom solutions.

At their most basic, Windows PowerShell will treat any valid Windows PowerShell script code saved in a `.psm1` file as a module. PowerShell will also automatically treat any binary cmdlet assembly as a module. However, you can also use a module (or more specifically, a module manifest) to bundle an entire solution together. The following scenarios describe typical uses for Windows PowerShell modules.

Libraries

Modules can be used to package and distribute cohesive libraries of functions that perform common tasks. Typically, the names of these functions share one or more nouns that reflect the common task that they are used for. These functions can also be similar to .NET Framework classes in that they can have public and private members. For example, a library can contain a set of functions for file transfers. In this case, the noun reflecting the common task might be "file."

Configuration

Modules can be used to customize your environment by adding specific cmdlets, providers, functions, and variables.

Compiled Code Development and Distribution

Cmdlet and provider developers can use modules to test and distribute their compiled code without needing to create snap-ins. They can import the assembly that contains the compiled code as a module (a binary module) without needing to create and register snap-ins.

See Also

[Understanding a Windows PowerShell Module](#)

[How to Write a PowerShell Script Module](#)

[How to Write a PowerShell Binary Module](#)

[How to Write a PowerShell Module Manifest](#)

[about_PSMODULEPATH](#)

[Importing a PowerShell Module](#)

[Installing a PowerShell Module](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Understanding a Windows PowerShell Module

Article • 09/17/2021

A *module* is a set of related Windows PowerShell functionalities, grouped together as a convenient unit (usually saved in a single directory). By defining a set of related script files, assemblies, and related resources as a module, you can reference, load, persist, and share your code much easier than you would otherwise.

The main purpose of a module is to allow the modularization (ie, reuse and abstraction) of Windows PowerShell code. For example, the most basic way of creating a module is to simply save a Windows PowerShell script as a `.psm1` file. Doing so allows you to control (ie, make public or private) the functions and variables contained in the script. Saving the script as a `.psm1` file also allows you to control the scope of certain variables. Finally, you can also use cmdlets such as [Install-Module](#) to organize, install, and use your script as building blocks for larger solutions.

Module Components and Types

A module is made up of four basic components:

1. Some sort of code file - usually either a PowerShell script or a managed cmdlet assembly.
2. Anything else that the above code file may need, such as additional assemblies, help files, or scripts.
3. A manifest file that describes the above files, as well as stores metadata such as author and versioning information.
4. A directory that contains all of the above content, and is located where PowerShell can reasonably find it.

ⓘ Note

None of these components, by themselves, are actually necessary. For example, a module can technically be only a script stored in a `.psm1` file. You can also have a module that is nothing but a manifest file, which is used mainly for organizational purposes. You can also write a script that dynamically creates a module, and as such doesn't actually need a directory to

store anything in. The following sections describe the types of modules you can get by mixing and matching the different possible parts of a module together.

Script Modules

As the name implies, a *script module* is a file (`.psm1`) that contains any valid Windows PowerShell code. Script developers and administrators can use this type of module to create modules whose members include functions, variables, and more. At heart, a script module is simply a Windows PowerShell script with a different extension, which allows administrators to use import, export, and management functions on it.

In addition, you can use a manifest file to include other resources in your module, such as data files, other dependent modules, or runtime scripts. Manifest files are also useful for tracking metadata such as authoring and versioning information.

Finally, a script module, like any other module that isn't dynamically created, needs to be saved in a folder that PowerShell can reasonably discover. Usually, this is on the PowerShell module path; but if necessary you can explicitly describe where your module is installed. For more information, see [How to Write a PowerShell Script Module](#).

Binary Modules

A **binary module** is a .NET Framework assembly (`.dll`) that contains compiled code, such as C#. Cmdlet developers can use this type of module to share cmdlets, providers, and more. (Existing snap-ins can also be used as binary modules.) Compared to a script module, a binary module allows you to create cmdlets that are faster or use features (such as multithreading) that are not as easy to code in Windows PowerShell scripts.

As with script modules, you can include a manifest file to describe additional resources that your module uses, and to track metadata about your module. Similarly, you probably should install your binary module in a folder somewhere along the PowerShell module path. For more information, see How to [How to Write a PowerShell Binary Module](#).

Manifest Modules

A **manifest module** is a module that uses a manifest file to describe all of its components, but doesn't have any sort of core assembly or script. (Formally, a manifest module leaves the `ModuleToProcess` or `RootModule` element of the manifest empty.) However, you can still use the other features of a module, such as the ability to load up

dependent assemblies or automatically run certain pre-processing scripts. You can also use a manifest module as a convenient way to package up resources that other modules will use, such as nested modules, assemblies, types, or formats. For more information, see [How to Write a PowerShell Module Manifest](#).

Dynamic Modules

A **dynamic module** is a module that is not loaded from, or saved to, a file. Instead, they are created dynamically by a script, using the [New-Module](#) cmdlet. This type of module enables a script to create a module on demand that does not need to be loaded or saved to persistent storage. By its nature, a dynamic module is intended to be short-lived, and therefore cannot be accessed by the [Get-Module](#) cmdlet. Similarly, they usually do not need module manifests, nor do they likely need permanent folders to store their related assemblies.

Module Manifests

A **module manifest** is a `.psd1` file that contains a hash table. The keys and values in the hash table do the following things:

- Describe the contents and attributes of the module.
- Define the prerequisites.
- Determine how the components are processed.

Manifests are not required for a module. Modules can reference script files (`.ps1`), script module files (`.psm1`), manifest files (`.psd1`), formatting and type files (`.ps1xml`), cmdlet and provider assemblies (`.dll`), resource files, Help files, localization files, or any other type of file or resource that is bundled as part of the module. For an internationalized script, the module folder also contains a set of message catalog files. If you add a manifest file to the module folder, you can reference the multiple files as a single unit by referencing the manifest.

The manifest itself describes the following categories of information:

- Metadata about the module, such as the module version number, the author, and the description.
- Prerequisites needed to import the module, such as the Windows PowerShell version, the common language runtime (CLR) version, and the required modules.
- Processing directives, such as the scripts, formats, and types to process.

- Restrictions on the members of the module to export, such as the aliases, functions, variables, and cmdlets to export.

For more information, see [How to Write a PowerShell Module Manifest](#).

Storing and Installing a Module

Once you have created a script, binary, or manifest module, you can save your work in a location that others may access it. For example, your module can be stored in the system folder where Windows PowerShell is installed, or it can be stored in a user folder.

Generally speaking, you can determine where you should install your module by using one of the paths stored in the `$Env:PSModulePath` variable. Using one of these paths means that PowerShell can automatically find and load your module when a user makes a call to it in their code. If you store your module somewhere else, you can explicitly let PowerShell know by passing in the location of your module as a parameter when you call `Install-Module`.

Regardless, the path of the folder is referred to as the **base** of the module (`ModuleBase`), and the name of the script, binary, or manifest module file should be the same as the module folder name, with the following exceptions:

- Dynamic modules that are created by the `New-Module` cmdlet can be named using the `Name` parameter of the cmdlet.
- Modules imported from assembly objects by the `Import-Module -Assembly` command are named according to the following syntax: `"dynamic_code_module_" + assembly.GetName()`.

For more information, see [Installing a PowerShell Module](#) and [about_PSMODULEPATH](#).

Module Cmdlets and Variables

The following cmdlets and variables are provided by Windows PowerShell for the creation and management of modules.

New-Module cmdlet This cmdlet creates a new dynamic module that exists only in memory. The module is created from a script block, and its exported members, such as its functions and variables, are immediately available in the session and remain available until the session is closed.

[New-ModuleManifest](#) cmdlet This cmdlet creates a new module manifest (`.psd1`) file, populates its values, and saves the manifest file to the specified path. This cmdlet can also be used to create a module manifest template that can be filled in manually.

[Import-Module](#) cmdlet This cmdlet adds one or more modules to the current session.

[Get-Module](#) cmdlet This cmdlet retrieves information about the modules that have been or that can be imported into the current session.

[Export-ModuleMember](#) cmdlet This cmdlet specifies the module members (such as cmdlets, functions, variables, and aliases) that are exported from a script module (`.psm1`) file or from a dynamic module created by using the [New-Module](#) cmdlet.

[Remove-Module](#) cmdlet This cmdlet removes modules from the current session.

[Test-ModuleManifest](#) cmdlet This cmdlet verifies that a module manifest accurately describes the components of a module by verifying that the files that are listed in the module manifest file (`.psd1`) actually exist in the specified paths.

`$PSScriptRoot` This variable contains the directory from which the script module is being executed. It enables scripts to use the module path to access other resources.

`$Env:PSModulePath` This environment variable contains a list of the directories in which Windows PowerShell modules are stored. Windows PowerShell uses the value of this variable when importing modules automatically and updating Help topics for modules.

See Also

[Writing a Windows PowerShell Module](#)

How to Write a PowerShell Script Module

Article • 06/09/2022

A script module is any valid PowerShell script saved in a `.psm1` extension. This extension allows the PowerShell engine to use rules and module cmdlets on your file. Most of these capabilities are there to help you install your code on other systems, as well as manage scoping. You can also use a module manifest file, which describes more complex installations and solutions.

Writing a PowerShell script module

To create a script module, save a valid PowerShell script to a `.psm1` file. The script and the directory where it's stored must use the same name. For example, a script named `MyPsScript.psm1` is stored in a directory named `MyPsScript`.

The module's directory needs to be in a path specified in `$Env:PSModulePath`. The module's directory can contain any resources that are needed to run the script, and a module manifest file that describes to PowerShell how your module works.

Create a basic PowerShell module

The following steps describe how to create a PowerShell module.

1. Save a PowerShell script with a `.psm1` extension. Use the same name for the script and the directory where the script is saved.

Saving a script with the `.psm1` extension means that you can use the module cmdlets, such as [Import-Module](#). The module cmdlets exist primarily so that you can import and export your code onto other user's systems. The alternate solution would be to load your code on other systems and then dot-source it into active memory, which isn't a scalable solution. For more information, see [Understanding a Windows PowerShell Module](#). By default, when users import your `.psm1` file, all functions in your script are accessible, but variables aren't.

An example PowerShell script, entitled `Show-Calendar`, is available at the end of this article.

```

function Show-Calendar {
    param(
        [datetime] $Start = [datetime]::Today,
        [datetime] $End = $Start,
        $FirstDayOfWeek,
        [int[]] $HighlightDay,
        [string[]] $HighlightDate = [datetime]::Today.ToString('yyyy-MM-
dd')
    )

    #actual code for the function goes here see the end of the topic
    #for the complete code sample
}

```

2. To control user access to certain functions or variables, call [Export-ModuleMember](#) at the end of your script.

The example code at the bottom of the article has only one function, which by default would be exposed. However, it's recommended you explicitly call out which functions you wish to expose, as described in the following code:

```

PowerShell

function Show-Calendar {
}
Export-ModuleMember -Function Show-Calendar

```

You can restrict what's imported using a module manifest. For more information, see [Importing a PowerShell Module](#) and [How to Write a PowerShell Module Manifest](#).

3. If you have modules that your own module needs to load, you can use [Import-Module](#), at the top of your module.

The [Import-Module](#) cmdlet imports a targeted module onto a system, and can be used at a later point in the procedure to install your own module. The sample code at the bottom of this article doesn't use any import modules. But if it did, they would be listed at the top of the file, as shown in the following code:

```

PowerShell

Import-Module GenericModule

```

4. To describe your module to the PowerShell Help system, you can either use standard help comments inside the file, or create an additional Help file.

The code sample at the bottom of this article includes the help information in the comments. You could also write expanded XML files that contain additional help content. For more information, see [Writing Help for Windows PowerShell Modules](#).

5. If you have additional modules, XML files, or other content you want to package with your module, you can use a module manifest.

A module manifest is a file that contains the names of other modules, directory layouts, versioning numbers, author data, and other pieces of information.

PowerShell uses the module manifest file to organize and deploy your solution. For more information, see [How to write a PowerShell module manifest](#).

6. To install and run your module, save the module to one of the appropriate PowerShell paths, and use `Import-Module`.

The paths where you can install your module are located in the `$Env:PSModulePath` global variable. For example, a common path to save a module on a system would be `%SystemRoot%/users/<user>/Documents/PowerShell/Modules/<moduleName>`. Be sure to create a directory for your module that uses the same name as the script module, even if it's only a single `.psm1` file. If you didn't save your module to one of these paths, you would have to specify the module's location in the `Import-Module` command. Otherwise, PowerShell wouldn't be able to find the module.

 **Note**

Starting with PowerShell 3.0, if you've placed your module in one of the PowerShell module paths, you don't need to explicitly import it. Your module is automatically loaded when a user calls your function. For more information about the module path, see [Importing a PowerShell Module](#) and [about_PSMODULEPATH](#).

7. To remove a module from active service in the current PowerShell session, use [Remove-Module](#).

 **Note**

`Remove-Module` removes a module from the current PowerShell session, but doesn't uninstall the module or delete the module's files.

Show-Calendar code example

The following example is a script module that contains a single function named `Show-Calendar`. This function displays a visual representation of a calendar. The sample contains the PowerShell Help strings for the synopsis, description, parameter values, and code. When the module is imported, the `Export-ModuleMember` command ensures that the `Show-Calendar` function is exported as a module member.

PowerShell

```
<#
.SYNOPSIS
    Displays a visual representation of a calendar.

.DESCRIPTION
    Displays a visual representation of a calendar. This function supports
    multiple months
    and lets you highlight specific date ranges or days.

.PARAMETER Start
    The first month to display.

.PARAMETER End
    The last month to display.

.PARAMETER FirstDayOfWeek
    The day of the month on which the week begins.

.PARAMETER HighlightDay
    Specific days (numbered) to highlight. Used for date ranges like (25..31).
    Date ranges are specified by the Windows PowerShell range syntax. These
    dates are
    enclosed in square brackets.

.PARAMETER HighlightDate
    Specific days (named) to highlight. These dates are surrounded by
    asterisks.

.EXAMPLE
    # Show a default display of this month.
    Show-Calendar

.EXAMPLE
    # Display a date range.
    Show-Calendar -Start "March, 2010" -End "May, 2010"

.EXAMPLE
    # Highlight a range of days.
    Show-Calendar -HighlightDay (1..10 + 22) -HighlightDate "2008-12-25"
#>
function Show-Calendar {
    param(
        [datetime] $Start = [datetime]::Today,
        [datetime] $End = $Start,
```

```

$FirstDayOfWeek,
[int[]] $HighlightDay,
[string[]] $HighlightDate = [datetime]::Today.ToString('yyyy-MM-dd')
)

## Determine the first day of the start and end months.
$Start = New-Object DateTime $Start.Year,$Start.Month,1
$End = New-Object DateTime $End.Year,$End.Month,1

## Convert the highlighted dates into real dates.
[datetime[]] $HighlightDate = [datetime[]] $HighlightDate

## Retrieve the DateTimeFormat information so that the
## calendar can be manipulated.
$dateTimeFormat = (Get-Culture).DateTimeFormat
if($FirstDayOfWeek)
{
    $dateTimeFormat.FirstDayOfWeek = $FirstDayOfWeek
}

$currentDay = $Start

## Process the requested months.
while($Start -le $End)
{
    ## Return to an earlier point in the function if the first day of the
    month
    ## is in the middle of the week.
    while($currentDay.DayOfWeek -ne $dateTimeFormat.FirstDayOfWeek)
    {
        $currentDay = $currentDay.AddDays(-1)
    }

    ## Prepare to store information about this date range.
    $currentWeek = New-Object PsObject
    $dayNames = @()
    $weeks = @()

    ## Continue processing dates until the function reaches the end of the
    month.
    ## The function continues until the week is completed with
    ## days from the next month.
    while(($currentDay -lt $Start.AddMonths(1)) -or
          ($currentDay.DayOfWeek -ne $dateTimeFormat.FirstDayOfWeek))
    {
        ## Determine the day names to use to label the columns.
        $dayName = "{0:ddd}" -f $currentDay
        if($dayNames -notcontains $dayName)
        {
            $dayNames += $dayName
        }

        ## Pad the day number for display, highlighting if necessary.
        $displayDay = " {0,2} " -f $currentDay.Day

```

```

## Determine whether to highlight a specific date.
if($HighlightDate)
{
    $compareDate = New-Object DateTime $currentDay.Year,
                  $currentDay.Month,$currentDay.Day
    if($HighlightDate -contains $compareDate)
    {
        $displayDay = "*" + ("{0,2}" -f $currentDay.Day) + "*"
    }
}

## Otherwise, highlight as part of a date range.
if($HighlightDay -and ($HighlightDay[0] -eq $currentDay.Day))
{
    $displayDay = "[" + ("{0,2}" -f $currentDay.Day) + "]"
    $null,$HighlightDay = $HighlightDay
}

## Add the day of the week and the day of the month as note
properties.
$currentWeek | Add-Member NoteProperty $dayName $displayDay

## Move to the next day of the month.
$currentDay = $currentDay.AddDays(1)

## If the function reaches the next week, store the current week
## in the week list and continue.
if($currentDay.DayOfWeek -eq $dateTimeFormat.FirstDayOfWeek)
{
    $weeks += $currentWeek
    $currentWeek = New-Object PsObject
}
}

## Format the weeks as a table.
$calendar = $weeks | Format-Table $dayNames -AutoSize | Out-String

## Add a centered header.
$width = ($calendar.Split("`n") | Measure-Object -Maximum
Length).Maximum
$header = "{0:MMMM yyyy}" -f $Start
$padding = " " * (($width - $header.Length) / 2)
$displayCalendar = "`n" + $padding + $header + "`n " + $calendar
$displayCalendar.TrimEnd()

## Move to the next month.
$Start = $Start.AddMonths(1)

}

}

Export-ModuleMember -Function Show-Calendar

```

How to Write a PowerShell Binary Module

06/12/2025

A binary module can be any assembly (.dll) that contains cmdlet classes. By default, all the cmdlets in the assembly are imported when the binary module is imported. However, you can restrict the cmdlets that are imported by creating a module manifest whose root module is the assembly. (For example, the **CmdletsToExport** key of the manifest can be used to export only those cmdlets that are needed.) In addition, a binary module can contain additional files, a directory structure, and other pieces of useful management information that a single cmdlet cannot.

The following procedure describes how to create and install a PowerShell binary module.

How to create and install a PowerShell binary module

1. Create a binary PowerShell solution (such as a cmdlet written in C#), with the capabilities you need, and ensure that it runs properly.

From a code perspective, the core of a binary module is a cmdlet assembly. In fact, PowerShell treats a single cmdlet assembly as a module for loading and unloading, with no additional effort on the part of the developer. For more information about writing a cmdlet, see [Writing a Windows PowerShell Cmdlet](#).

2. If necessary, create the rest of your solution: (additional cmdlets, XML files, and so on) and describe them with a module manifest.

In addition to describing the cmdlet assemblies in your solution, a module manifest can describe how you want your module exported and imported, what cmdlets will be exposed, and what additional files will go into the module. As stated previously however, PowerShell can treat a binary cmdlet like a module with no additional effort. As such, a module manifest is useful mainly for combining multiple files into a single package, or for explicitly controlling publication for a given assembly. For more information, see [How to Write a PowerShell Module Manifest](#).

The following code is a simplified C# example that contains three cmdlets in the same file that can be used as a module.

C#

```
using System.Management.Automation;           // Windows PowerShell
namespace.
```

```

namespace ModuleCmdlets
{
    [Cmdlet(VerbsDiagnostic.Test, "BinaryModuleCmdlet1")]
    public class TestBinaryModuleCmdlet1Command : Cmdlet
    {
        protected override void BeginProcessing()
        {
            WriteObject("BinaryModuleCmdlet1 exported by the ModuleCmdlets
module.");
        }
    }

    [Cmdlet(VerbsDiagnostic.Test, "BinaryModuleCmdlet2")]
    public class TestBinaryModuleCmdlet2Command : Cmdlet
    {
        protected override void BeginProcessing()
        {
            WriteObject("BinaryModuleCmdlet2 exported by the ModuleCmdlets
module.");
        }
    }

    [Cmdlet(VerbsDiagnostic.Test, "BinaryModuleCmdlet3")]
    public class TestBinaryModuleCmdlet3Command : Cmdlet
    {
        protected override void BeginProcessing()
        {
            WriteObject("BinaryModuleCmdlet3 exported by the ModuleCmdlets
module.");
        }
    }
}

```

3. Package your solution, and save the package to somewhere in the PowerShell module path.

The `$env:PSModulePath` global environment variable describes the default paths that PowerShell uses to locate your module. For example, a common path to save a module on a system would be `%SystemRoot%\Users\<user>\Documents\WindowsPowerShell\Modules\<moduleName>`. If you don't use the default paths, you need to explicitly state the location of your module during installation. Be sure to create a folder to save your module in, as you may need the folder to store multiple assemblies and files for your solution.

Technically, you don't need to install your module anywhere on the `$env:PSModulePath` - those are simply the default locations that PowerShell will look for your module. However, it's considered best practice to do so, unless you have a good reason for storing your

module somewhere else. For more information, see [Installing a PowerShell Module](#) and [about_PSMODULEPATH](#).

4. Import your module into PowerShell with a call to [Import-Module](#).

Calling to [Import-Module](#) loads your module into active memory. If you are using PowerShell 3.0 and later, invoking a command from your module in code also imports it. For more information, see [Importing a PowerShell Module](#).

Module initialization and cleanup code

If your module needs to do something upon import or removal such as a discovery task or initialization, you can implement the [IModuleAssemblyInitializer](#) and [IModuleAssemblyCleanup](#) interfaces.

 Note

This pattern is discouraged unless absolutely necessary. To keep PowerShell performant, you should lazily load things at the point your commands are called rather than on import.

Importing snap-in assemblies as modules

Cmdlets and providers that exist in snap-in assemblies can be loaded as binary modules. When the snap-in assemblies are loaded as binary modules, the cmdlets and providers in the snap-in are available to the user, but the snap-in class in the assembly is ignored, and the snap-in isn't registered. As a result, the snap-in cmdlets provided by Windows PowerShell can't detect the snap-in even though the cmdlets and providers are available to the session.

In addition, any formatting or types files that are referenced by the snap-in can't be imported as part of a binary module. To import the formatting and types files you must create a module manifest. See, [How to Write a PowerShell Module Manifest](#).

See Also

- [Writing a Windows PowerShell Module](#)

How to write a PowerShell module manifest

Article • 02/06/2024

After you've written your PowerShell module, you can add an optional module manifest that includes information about the module. For example, you can describe the author, specify files in the module (such as nested modules), run scripts to customize the user's environment, load type and formatting files, define system requirements, and limit the members that the module exports.

Creating a module manifest

A **module manifest** is a PowerShell data file (`.psd1`) that describes the contents of a module and determines how a module is processed. The manifest file is a text file that contains a hash table of keys and values. You link a manifest file to a module by naming the manifest the same as the module, and storing the manifest in the module's root directory.

For simple modules that contain only a single `.psm1` or binary assembly, a module manifest is optional. But, the recommendation is to use a module manifest whenever possible, as they're useful to help you organize your code and maintain versioning information. And, a module manifest is required to export an assembly that is installed in the [Global Assembly Cache](#). A module manifest is also required for modules that support the Updatable Help feature. Updatable Help uses the **HelpInfoUri** key in the module manifest to find the Help information (HelpInfo XML) file that contains the location of the updated help files for the module. For more information about Updatable Help, see [Supporting Updatable Help](#).

To create and use a module manifest

1. The best practice to create a module manifest is to use the [New-ModuleManifest](#) cmdlet. You can use parameters to specify one or more of the manifest's default keys and values. The only requirement is to name the file. `New-ModuleManifest` creates a module manifest with your specified values, and includes the remaining keys and their default values. If you need to create multiple modules, use `New-ModuleManifest` to create a module manifest template that can be modified for your different modules. For an example of a default module manifest, see the [Sample module manifest](#).

```
New-ModuleManifest -Path C:\myModuleName.psd1 -ModuleVersion "2.0" -Author  
"YourNameHere"
```

An alternative is to manually create the module manifest's hash table using the minimal information required, the **ModuleVersion**. You save the file with the same name as your module and use the `.psd1` file extension. You can then edit the file and add the appropriate keys and values.

2. Add any additional elements that you want in the manifest file.

To edit the manifest file, use any text editor you prefer. But, the manifest file is a script file that contains code, so you may wish to edit it in a scripting or development environment, such as Visual Studio Code. All elements of a manifest file are optional, except for the **ModuleVersion** number.

For descriptions of the keys and values you can include in a module manifest, see the [Module manifest elements](#) table. For more information, see the parameter descriptions in the [New-ModuleManifest](#) cmdlet.

3. To address any scenarios that might not be covered by the base module manifest elements, you have the option to add additional code to your module manifest.

For security concerns, PowerShell only runs a small subset of the available operations in a module manifest file. Generally, you can use the `if` statement, arithmetic and comparison operators, and the basic PowerShell data types.

4. After you've created your module manifest, you can test it to confirm that any paths described in the manifest are correct. To test your module manifest, use [Test-ModuleManifest](#).

```
Test-ModuleManifest myModuleName.psd1
```

5. Be sure that your module manifest is located in the top level of the directory that contains your module.

When you copy your module onto a system and import it, PowerShell uses the module manifest to import your module.

6. Optionally, you can directly test your module manifest with a call to [Import-Module](#) by dot-sourcing the manifest itself.

```
Import-Module .\myModuleName.psd1
```

Module manifest elements

The following table describes the elements you can include in a module manifest.

[+] Expand table

Element	Default	Description
RootModule Type: <code>String</code>	<code><empty string></code>	<p>Script module or binary module file associated with this manifest. Previous versions of PowerShell called this element the ModuleToProcess.</p> <p>Possible types for the root module can be empty, which creates a Manifest module, the name of a script module (<code>.psm1</code>), or the name of a binary module (<code>.exe</code> or <code>.d11</code>). Placing the name of a module manifest (<code>.psd1</code>) or a script file (<code>.ps1</code>) in this element causes an error.</p> <p>Example: <code>RootModule = 'ScriptModule.psm1'</code></p>
ModuleVersion Type: <code>Version</code>	<code>'0.0.1'</code>	<p>Version number of this module. If a value isn't specified, <code>New-ModuleManifest</code> uses the default. The string must be able to convert to the type <code>Version</code> for example <code>#.#.#.#</code>. <code>Import-Module</code> loads the first module it finds on the <code>\$PSModulePath</code> that matches the name, and has at least as high a ModuleVersion, as the MinimumVersion parameter. To import a specific version, use the <code>Import-Module</code> cmdlet's RequiredVersion parameter.</p> <p>Example: <code>ModuleVersion = '1.0'</code></p>
GUID Type: <code>GUID</code>	<code>'<GUID>'</code>	<p>ID used to uniquely identify this module. If a value isn't specified, <code>New-ModuleManifest</code> autogenerates the value. You can't currently import a module by GUID.</p> <p>Example: <code>GUID = 'cfc45206-1e49-459d-a8ad-5b571ef94857'</code></p>
Author Type: <code>String</code>	<code>'<Current user>'</code>	<p>Author of this module. If a value isn't specified, <code>New-ModuleManifest</code> uses the current user.</p> <p>Example: <code>Author = 'AuthorNameHere'</code></p>
CompanyName Type: <code>String</code>	<code>'Unknown'</code>	<p>Company or vendor of this module. If a value isn't specified, <code>New-ModuleManifest</code> uses the default.</p> <p>Example: <code>CompanyName = 'Fabrikam'</code></p>
Copyright Type: <code>String</code>	<code>'(c)<Author>. All rights reserved.'</code>	<p>Copyright statement for this module. If a value isn't specified, <code>New-ModuleManifest</code> uses the default with the current user as the <code><Author></code>. To specify an author, use the Author parameter.</p>

Element	Default	Description
		Example: <code>Copyright = '2019 AuthorName. All rights reserved.'</code>
Description Type: <code>String</code>	<code><empty string></code>	<p>Description of the functionality provided by this module.</p> <p>Example: <code>Description = 'This is the module's description.'</code></p>
PowerShellVersion Type: <code>Version</code>	<code><empty string></code>	<p>Minimum version of the PowerShell engine required by this module. Valid values are 1.0, 2.0, 3.0, 4.0, 5.0, 5.1, 6.0, 6.1, 6.2, 7.0 and 7.1.</p> <p>Example: <code>PowerShellVersion = '5.0'</code></p>
PowerShellHostName Type: <code>String</code>	<code><empty string></code>	<p>Name of the PowerShell host required by this module. This name is provided by PowerShell. To find the name of a host program, in the program, type: <code>\$Host.Name</code>.</p> <p>Example: <code>PowerShellHostName = 'ConsoleHost'</code></p>
PowerShellHostVersion Type: <code>Version</code>	<code><empty string></code>	<p>Minimum version of the PowerShell host required by this module.</p> <p>Example: <code>PowerShellHostVersion = '2.0'</code></p>
DotNetFrameworkVersion Type: <code>Version</code>	<code><empty string></code>	<p>Minimum version of Microsoft .NET Framework required by this module. This prerequisite is valid for the PowerShell Desktop edition only, such as Windows PowerShell 5.1, and only applies to .NET Framework versions lower than 4.5.</p> <p>Example: <code>DotNetFrameworkVersion = '3.5'</code></p>
CLRVersion Type: <code>Version</code>	<code><empty string></code>	<p>Minimum version of the common language runtime (CLR) required by this module. This prerequisite is valid for the PowerShell Desktop edition only, such as Windows PowerShell 5.1, and only applies to .NET Framework versions lower than 4.5.</p> <p>Example: <code>CLRVersion = '3.5'</code></p>
ProcessorArchitecture Type: <code>ProcessorArchitecture</code>	<code><empty string></code>	<p>Processor architecture (None, X86, Amd64) required by this module. Valid values are x86, AMD64, Arm, IA64, MSIL, and None (unknown or unspecified).</p> <p>Example: <code>ProcessorArchitecture = 'x86'</code></p>
RequiredModules Type: <code>Object[]</code>	<code>@()</code>	<p>Modules that must be imported into the global environment prior to importing this module. This loads any modules listed unless they've already been loaded. For example, some modules may already be loaded by a different module. It's</p>

Element	Default	Description
		<p>possible to specify a specific version to load using <code>RequiredVersion</code> rather than <code>ModuleVersion</code>. When <code>ModuleVersion</code> is used it will load the newest version available with a minimum of the version specified. You can combine strings and hash tables in the parameter value.</p> <p>Example: <code>RequiredModules = @("MyModule", @{ModuleName="MyDependentModule"; ModuleVersion="2.0"; GUID="cfc45206-1e49-459d-a8ad-5b571ef94857"})</code></p> <p>Example: <code>RequiredModules = @("MyModule", @{ModuleName="MyDependentModule"; RequiredVersion="1.5"; GUID="cfc45206-1e49-459d-a8ad-5b571ef94857"})</code></p>
RequiredAssemblies	@()	<p>Assemblies that must be loaded prior to importing this module. Specifies the assembly (.dll) file names that the module requires.</p> <p>PowerShell loads the specified assemblies before updating types or formats, importing nested modules, or importing the module file that is specified in the value of the <code>RootModule</code> key. Use this parameter to list all the assemblies that the module requires.</p> <p>Example: <code>RequiredAssemblies = @("assembly1.dll", "assembly2.dll", "assembly3.dll")</code></p>
ScriptsToProcess	@()	<p>Script (.ps1) files that are run in the caller's session state when the module is imported. This could be the global session state or, for nested modules, the session state of another module. You can use these scripts to prepare an environment just as you might use a log in script.</p> <p>These scripts are run before any of the modules listed in the manifest are loaded.</p> <p>Example: <code>ScriptsToProcess = @("script1.ps1", "script2.ps1", "script3.ps1")</code></p>
TypesToProcess	@()	<p>Type files (.ps1xml) to be loaded when importing this module.</p> <p>Example: <code>TypesToProcess = @("type1.ps1xml", "type2.ps1xml", "type3.ps1xml")</code></p>
FormatsToProcess	@()	<p>Format files (.ps1xml) to be loaded when importing this module.</p>

Element	Default	Description
		<p>Example: <code>FormatsToProcess = @("format1.ps1xml", "format2.ps1xml", "format3.ps1xml")</code></p>
NestedModules Type: <code>Object[]</code>	<code>@()</code>	<p>Modules to import as nested modules of the module specified in RootModule (alias:ModuleToProcess). Adding a module name to this element is similar to calling <code>Import-Module</code> from within your script or assembly code. The main difference by using a manifest file is that it's easier to see what you're loading. And, if a module fails to load, you will not yet have loaded your actual module.</p> <p>In addition to other modules, you may also load script (<code>.ps1</code>) files here. These files will execute in the context of the root module. This is equivalent to dot sourcing the script in your root module.</p> <p>Example: <code>NestedModules = @("script.ps1", @{ModuleName="MyModule"; ModuleVersion="1.0.0.0"; GUID="50cdb55f-5ab7-489f-9e94-4ec21ff51e59"})</code></p>
FunctionsToExport Type: <code>String[]</code>	<code>@()</code>	<p>Specifies the functions to export from this module, for best performance, do not use wildcards and do not delete the entry, use an empty array if there are no functions to export. By default, no functions are exported. You can use this key to list the functions that are exported by the module.</p> <p>The module exports the functions to the caller's session state. The caller's session state can be the global session state or, for nested modules, the session state of another module. When chaining nested modules, all functions that are exported by a nested module will be exported to the global session state unless a module in the chain restricts the function by using the FunctionsToExport key. If the manifest exports aliases for the functions, this key can remove functions whose aliases are listed in the AliasesToExport key, but this key cannot add function aliases to the list.</p> <p>Example: <code>FunctionsToExport = @("function1", "function2", "function3")</code></p>
CmdletsToExport Type: <code>String[]</code>	<code>@()</code>	<p>Specifies the cmdlets to export from this module, for best performance, do not use wildcards and do not delete the entry, use an empty array if there are no cmdlets to export. By default, no cmdlets are exported. You can use this key to list the cmdlets that are exported by the module.</p>

Element	Default	Description
		<p>The caller's session state can be the global session state or, for nested modules, the session state of another module. When you're chaining nested modules, all cmdlets that are exported by a nested module will be exported to the global session state unless a module in the chain restricts the cmdlet by using the CmdletsToExport key.</p> <p>If the manifest exports aliases for the cmdlets, this key can remove cmdlets whose aliases are listed in the AliasesToExport key, but this key cannot add cmdlet aliases to the list.</p> <p>Example: <code>CmdletsToExport = @("Get-MyCmdlet", "Set-MyCmdlet", "Test-MyCmdlet")</code></p>
VariablesToExport Type: <code>String[]</code>	<code>'*'</code>	<p>Specifies the variables that the module exports to the caller's session state. Wildcard characters are permitted. By default, all variables (<code>'*'</code>) are exported. You can use this key to restrict the variables that are exported by the module.</p> <p>The caller's session state can be the global session state or, for nested modules, the session state of another module. When you are chaining nested modules, all variables that are exported by a nested module will be exported to the global session state unless a module in the chain restricts the variable by using the VariablesToExport key.</p> <p>If the manifest also exports aliases for the variables, this key can remove variables whose aliases are listed in the AliasesToExport key, but this key cannot add variable aliases to the list.</p> <p>Example: <code>VariablesToExport = @('\$MyVariable1', '\$MyVariable2', '\$MyVariable3')</code></p>
AliasesToExport Type: <code>String[]</code>	<code>@()</code>	<p>Specifies the aliases to export from this module, for best performance, do not use wildcards and do not delete the entry, use an empty array if there are no aliases to export. By default, no aliases are exported. You can use this key to list the aliases that are exported by the module.</p> <p>The module exports the aliases to caller's session state. The caller's session state can be the global session state or, for nested modules, the session state of another module. When you are chaining nested modules, all aliases that are exported by a nested module will be ultimately exported to the global session state unless a module in the chain restricts the alias by using the AliasesToExport key.</p>

Element	Default	Description
		Example: <code>AliasesToExport = @("MyAlias1", "MyAlias2", "MyAlias3")</code>
DscResourcesToExport Type: <code>String[]</code>	<code>@()</code>	Specifies DSC resources to export from this module. Wildcards are permitted. Example: <code>DscResourcesToExport = @("DscResource1", "DscResource2", "DscResource3")</code>
ModuleList Type: <code>Object[]</code>	<code>@()</code>	Specifies all the modules that are packaged with this module. These modules can be entered by name, using a comma-separated string, or as a hash table with ModuleName and GUID keys. The hash table can also have an optional ModuleVersion key. The ModuleList key is designed to act as a module inventory. These modules are not automatically processed. Example: <code>ModuleList = @("SampleModule", "MyModule", @{ModuleName="MyModule"; ModuleVersion="1.0.0.0"; GUID="50cdb55f-5ab7-489f-9e94-4ec21ff51e59"})</code>
FileList Type: <code>String[]</code>	<code>@()</code>	List of all files packaged with this module. As with ModuleList , FileList is an inventory list, and isn't otherwise processed. Example: <code>FileList = @("File1", "File2", "File3")</code>
PrivateData Type: <code>Object</code>	<code>@{...}</code>	Specifies any private data that needs to be passed to the root module specified by the RootModule (alias: ModuleToProcess) key. PrivateData is a hash table that comprises several elements: Tags , LicenseUri , ProjectURI , IconURI , ReleaseNotes , Prerelease , RequireLicenseAcceptance , and ExternalModuleDependencies .
Tags Type: <code>String[]</code>	<code>@()</code>	Tags help with module discovery in online galleries. Example: <code>Tags = "PackageManagement", "PowerShell", "Manifest"</code>
LicenseUri Type: <code>Uri</code>	<code><empty string></code>	A URL to the license for this module. Example: <code>LicenseUri = 'https://www.contoso.com/license'</code>
ProjectUri Type: <code>Uri</code>	<code><empty string></code>	A URL to the main website for this project. Example: <code>ProjectUri = 'https://www.contoso.com/project'</code>

Element	Default	Description
IconUri Type: Uri	<empty string>	A URL to an icon representing this module. Example: IconUri = ' https://www.contoso.com/icons/icon.png '
ReleaseNotes Type: String	<empty string>	Specifies the module's release notes. Example: ReleaseNotes = 'The release notes provide information about the module.'
PreRelease Type: String	<empty string>	This parameter was added in PowerShellGet 1.6.6. A PreRelease string that identifies the module as a prerelease version in online galleries. Example: PreRelease = 'alpha'
RequireLicenseAcceptance Type: Boolean	\$false	This parameter was added in PowerShellGet 1.5. Flag to indicate whether the module requires explicit user acceptance for install, update, or save. Example: RequireLicenseAcceptance = \$false
ExternalModuleDependencies Type: String[]	@()	This parameter was added in PowerShellGet v2. A list of external modules that this module is dependent upon. Example: ExternalModuleDependencies = @("ExtModule1", "ExtModule2", "ExtModule3")
HelpInfoURI Type: String	<empty string>	HelpInfo URI of this module. Example: HelpInfoURI = ' https://www.contoso.com/help '
DefaultCommandPrefix Type: String	<empty string>	Default prefix for commands exported from this module. Override the default prefix using Import-Module -Prefix. Example: DefaultCommandPrefix = 'My'

Sample module manifest

The following sample module manifest was created with `New-ModuleManifest` in PowerShell 7 and contains the default keys and values.

```
PowerShell

#
# Module manifest for module 'SampleModuleManifest'
#
# Generated by: User01
#
# Generated on: 10/15/2019
```

```
#  
  
#{@  
  
# Script module or binary module file associated with this manifest.  
# RootModule = ''  
  
# Version number of this module.  
ModuleVersion = '0.0.1'  
  
# Supported PSEditions  
# CompatiblePSEditions = @()  
  
# ID used to uniquely identify this module  
GUID = 'b632e90c-df3d-4340-9f6c-3b832646bf87'  
  
# Author of this module  
Author = 'User01'  
  
# Company or vendor of this module  
CompanyName = 'Unknown'  
  
# Copyright statement for this module  
Copyright = '(c) User01. All rights reserved.'  
  
# Description of the functionality provided by this module  
# Description = ''  
  
# Minimum version of the PowerShell engine required by this module  
# PowerShellVersion = ''  
  
# Name of the PowerShell host required by this module  
# PowerShellHostName = ''  
  
# Minimum version of the PowerShell host required by this module  
# PowerShellHostVersion = ''  
  
# Minimum version of Microsoft .NET Framework required by this module. This  
# prerequisite is valid for the PowerShell Desktop edition only.  
# DotNetFrameworkVersion = ''  
  
# Minimum version of the common language runtime (CLR) required by this  
# module. This prerequisite is valid for the PowerShell Desktop edition only.  
# CLRVersion = ''  
  
# Processor architecture (None, X86, Amd64) required by this module  
# ProcessorArchitecture = ''  
  
# Modules that must be imported into the global environment prior to  
# importing this module  
# RequiredModules = @()  
  
# Assemblies that must be loaded prior to importing this module  
# RequiredAssemblies = @()
```

```
# Script files (.ps1) that are run in the caller's environment prior to
# importing this module.
# ScriptsToProcess = @()

# Type files (.ps1xml) to be loaded when importing this module
# TypesToProcess = @()

# Format files (.ps1xml) to be loaded when importing this module
# FormatsToProcess = @()

# Modules to import as nested modules of the module specified in
# RootModule/ModuleToProcess
# NestedModules = @()

# Functions to export from this module, for best performance, do not use
# wildcards and do not delete the entry, use an empty array if there are no
# functions to export.
FunctionsToExport = @()

# Cmdlets to export from this module, for best performance, do not use
# wildcards and do not delete the entry, use an empty array if there are no
# cmdlets to export.
CmdletsToExport = @()

# Variables to export from this module
VariablesToExport = '*'

# Aliases to export from this module, for best performance, do not use
# wildcards and do not delete the entry, use an empty array if there are no
# aliases to export.
AliasesToExport = @()

# DSC resources to export from this module
# DscResourcesToExport = @()

# List of all modules packaged with this module
# ModuleList = @()

# List of all files packaged with this module
# FileList = @()

# Private data to pass to the module specified in
# RootModule/ModuleToProcess. This may also contain a PSData hashtable with
# additional module metadata used by PowerShell.
PrivateData = @{

    PSData = @{

        # Tags applied to this module. These help with module discovery in
        # online galleries.
        # Tags = @()

        # A URL to the license for this module.
        # LicenseUri = ''
    }
}
```

```
# A URL to the main website for this project.  
# ProjectUri = ''  
  
# A URL to an icon representing this module.  
# IconUri = ''  
  
# ReleaseNotes of this module  
# ReleaseNotes = ''  
  
# Prerelease string of this module  
# Prerelease = ''  
  
# Flag to indicate whether the module requires explicit user  
acceptance for install/update/save  
# RequireLicenseAcceptance = $false  
  
# External dependent modules of this module  
# ExternalModuleDependencies = @()  
  
} # End of PSData hashtable  
  
} # End of PrivateData hashtable  
  
# HelpInfo URI of this module  
# HelpInfoURI = ''  
  
# Default prefix for commands exported from this module. Override the  
default prefix using Import-Module -Prefix.  
# DefaultCommandPrefix = ''  
  
}
```

See also

- [about_Comparison_Operators](#)
- [about_If](#)
- [Global Assembly Cache](#)
- [Import-Module](#)
- [New-ModuleManifest](#)
- [Test-ModuleManifest](#)
- [Update-ModuleManifest](#)
- [Writing a Windows PowerShell Module](#)

Installing a PowerShell Module

Article • 11/15/2022

After you have created your PowerShell module, you will likely want to install the module on a system, so that you or others may use it. Generally speaking, this consists of copying the module files (ie, the `.psm1`, or the binary assembly, the module manifest, and any other associated files) onto a directory on that computer. For a very small project, this may be as simple as copying and pasting the files with Windows Explorer onto a single remote computer; however, for larger solutions you may wish to use a more sophisticated installation process. Regardless of how you get your module onto the system, PowerShell can use a number of techniques that will let users find and use your modules. Therefore, the main issue for installation is ensuring that PowerShell will be able to find your module. For more information, see [Importing a PowerShell Module](#).

Rules for Installing Modules

The following information pertains to all modules, including modules that you create for your own use, modules that you get from other parties, and modules that you distribute to others.

Install Modules in PSModulePath

Whenever possible, install all modules in a path that is listed in the **PSModulePath** environment variable or add the module path to the **PSModulePath** environment variable value.

The **PSModulePath** environment variable (`$Env:PSModulePath`) contains the locations of Windows PowerShell modules. Cmdlets rely on the value of this environment variable to find modules.

By default, the **PSModulePath** environment variable value contains the following system and user module directories, but you can add to and edit the value.

- `$PSHOME\Modules` (`%windir%\System32\WindowsPowerShell\v1.0\Modules`)

Warning

This location is reserved for modules that ship with Windows. Do not install modules to this location.

- `$HOME\Documents\WindowsPowerShell\Modules`
`(%HOMEDRIVE%%HOMEPATH%\Documents\WindowsPowerShell\Modules)`
- `$Env:ProgramFiles\WindowsPowerShell\Modules`
`(%ProgramFiles%\WindowsPowerShell\Modules)`

To get the value of the **PSModulePath** environment variable, use either of the following commands.

PowerShell

```
$Env:PSModulePath  
[Environment]::GetEnvironmentVariable("PSModulePath")
```

To add a module path to value of the **PSModulePath** environment variable value, use the following command format. This format uses the **SetEnvironmentVariable** method of the **System.Environment** class to make a session-independent change to the **PSModulePath** environment variable.

PowerShell

```
#Save the current value in the $p variable.  
$p = [Environment]::GetEnvironmentVariable("PSModulePath")  
  
#Add the new path to the $p variable. Begin with a semi-colon  
#separator.  
$p += ";C:\Program Files (x86)\MyCompany\Modules\"  
  
#Add the paths in $p to the PSModulePath value.  
[Environment]::SetEnvironmentVariable("PSModulePath",$p)
```

Important

Once you have added the path to **PSModulePath**, you should broadcast an environment message about the change. Broadcasting the change allows other applications, such as the shell, to pick up the change. To broadcast the change, have your product installation code send a **WM_SETTINGCHANGE** message with `lParam` set to the string "Environment". Be sure to send the message after your module installation code has updated **PSModulePath**.

Use the Correct Module Directory Name

A well-formed module is a module that is stored in a directory that has the same name as the base name of at least one file in the module directory. If a module is not well-formed, Windows PowerShell does not recognize it as a module.

The "base name" of a file is the name without the file name extension. In a well-formed module, the name of the directory that contains the module files must match the base name of at least one file in the module.

For example, in the sample Fabrikam module, the directory that contains the module files is named "Fabrikam" and at least one file has the "Fabrikam" base name. In this case, both Fabrikam.psd1 and Fabrikam.dll have the "Fabrikam" base name.

```
C:\Program Files
  Fabrikam Technologies
    Fabrikam Manager
      Modules
        Fabrikam
          Fabrikam.psd1 (module manifest)
          Fabrikam.dll (module assembly)
```

Effect of Incorrect Installation

If the module is not well-formed and its location is not included in the value of the **PSModulePath** environment variable, basic discovery features of Windows PowerShell, such as the following, do not work.

- The Module Auto-Loading feature cannot import the module automatically.
- The `ListAvailable` parameter of the [Get-Module](#) cmdlet cannot find the module.
- The [Import-Module](#) cmdlet cannot find the module. To import the module, you must provide the full path to the root module file or module manifest file.

Additional features, such as the following, do not work unless the module is imported into the session. In well-formed modules in the **PSModulePath** environment variable, these features work even when the module is not imported into the session.

- The [Get-Command](#) cmdlet cannot find commands in the module.
- The [Update-Help](#) and [Save-Help](#) cmdlets cannot update or save help for the module.

- The `Show-Command` cmdlet cannot find and display the commands in the module.

The commands in the module are missing from the `Show-Command` window in Windows PowerShell Integrated Scripting Environment (ISE).

Where to Install Modules

This section explains where in the file system to install Windows PowerShell modules. The location depends on how the module is used.

Installing Modules for a Specific User

If you create your own module or get a module from another party, such as a Windows PowerShell community website, and you want the module to be available for your user account only, install the module in your user-specific Modules directory.

```
$HOME\Documents\WindowsPowerShell\Modules\<Module Folder>\<Module Files>
```

The user-specific Modules directory is added to the value of the `PSModulePath` environment variable by default.

Installing Modules for all Users in Program Files

If you want a module to be available to all user accounts on the computer, install the module in the Program Files location.

```
$Env:ProgramFiles\WindowsPowerShell\Modules\<Module Folder>\<Module Files>
```

ⓘ Note

The Program Files location is added to the value of the `PSModulePath` environment variable by default in Windows PowerShell 4.0 and later. For earlier versions of Windows PowerShell, you can manually create the Program Files location (`%ProgramFiles%\WindowsPowerShell\Modules`) and add this path to your `PSModulePath` environment variable as described above.

Installing Modules in a Product Directory

If you are distributing the module to other parties, use the default Program Files location described above, or create your own company-specific or product-specific subdirectory of the `%ProgramFiles%` directory.

For example, Fabrikam Technologies, a fictitious company, is shipping a Windows PowerShell module for their Fabrikam Manager product. Their module installer creates a Modules subdirectory in the Fabrikam Manager product subdirectory.

```
C:\Program Files  
  Fabrikam Technologies  
    Fabrikam Manager  
      Modules  
        Fabrikam  
          Fabrikam.psd1 (module manifest)  
          Fabrikam.dll (module assembly)
```

To enable the Windows PowerShell module discovery features to find the Fabrikam module, the Fabrikam module installer adds the module location to the value of the **PSModulePath** environment variable.

```
PowerShell  
  
$p = [Environment]::GetEnvironmentVariable("PSModulePath")  
$p += ";C:\Program Files\Fabrikam Technologies\Fabrikam Manager\Modules\"  
[Environment]::SetEnvironmentVariable("PSModulePath",$p)
```

Installing Modules in the Common Files Directory

If a module is used by multiple components of a product or by multiple versions of a product, install the module in a module-specific subdirectory of the `%ProgramFiles%\Common Files\Modules` subdirectory.

In the following example, the Fabrikam module is installed in a Fabrikam subdirectory of the `%ProgramFiles%\Common Files\Modules` subdirectory. Note that each module resides in its own subdirectory in the Modules subdirectory.

```
C:\Program Files  
  Common Files  
    Modules  
      Fabrikam  
        Fabrikam.psd1 (module manifest)  
        Fabrikam.dll (module assembly)
```

Then, the installer assures the value of the **PSModulePath** environment variable includes the path of the `Common Files\Modules` subdirectory.

PowerShell

```
$m = $Env:ProgramFiles + '\Common Files\Modules'  
$p = [Environment]::GetEnvironmentVariable("PSModulePath")  
$q = $p -split ';'  
if ($q -notcontains $m) {  
    $q += ";$m"  
}  
$p = $q -join ';'  
[Environment]::SetEnvironmentVariable("PSModulePath", $p)
```

Installing Multiple Versions of a Module

To install multiple versions of the same module, use the following procedure.

1. Create a directory for each version of the module. Include the version number in the directory name.
2. Create a module manifest for each version of the module. In the value of the **ModuleVersion** key in the manifest, enter the module version number. Save the manifest file (.psd1) in the version-specific directory for the module.
3. Add the module root folder path to the value of the **PSModulePath** environment variable, as shown in the following examples.

To import a particular version of the module, the end-user can use the `MinimumVersion` or `RequiredVersion` parameters of the [Import-Module](#) cmdlet.

For example, if the Fabrikam module is available in versions 8.0 and 9.0, the Fabrikam module directory structure might resemble the following.

```
C:\Program Files  
Fabrikam Manager  
  Fabrikam8  
    Fabrikam  
      Fabrikam.psd1 (module manifest: ModuleVersion = "8.0")  
      Fabrikam.dll (module assembly)  
  Fabrikam9  
    Fabrikam  
      Fabrikam.psd1 (module manifest: ModuleVersion = "9.0")  
      Fabrikam.dll (module assembly)
```

The installer adds both of the module paths to the **PSModulePath** environment variable value.

PowerShell

```
$p = [Environment]::GetEnvironmentVariable("PSModulePath")
$p += ";C:\Program Files\Fabrikam\Fabrikam8;C:\Program
Files\Fabrikam\Fabrikam9"
[Environment]::SetEnvironmentVariable("PSModulePath",$p)
```

When these steps are complete, the [ListAvailable](#) parameter of the [Get-Module](#) cmdlet gets both of the Fabrikam modules. To import a particular module, use the [MinimumVersion](#) or [RequiredVersion](#) parameters of the [Import-Module](#) cmdlet.

If both modules are imported into the same session, and the modules contain cmdlets with the same names, the cmdlets that are imported last are effective in the session.

Handling Command Name Conflicts

Command name conflicts can occur when the commands that a module exports have the same name as commands in the user's session.

When a session contains two commands that have the same name, Windows PowerShell runs the command type that takes precedence. When a session contains two commands that have the same name and the same type, Windows PowerShell runs the command that was added to the session most recently. To run a command that is not run by default, users can qualify the command name with the module name.

For example, if the session contains a [Get-Date](#) function and the [Get-Date](#) cmdlet, Windows PowerShell runs the function by default. To run the cmdlet, preface the command with the module name, such as:

PowerShell

```
Microsoft.PowerShell.Utility\Get-Date
```

To prevent name conflicts, module authors can use the [DefaultCommandPrefix](#) key in the module manifest to specify a noun prefix for all commands exported from the module.

Users can use the [Prefix](#) parameter of the [Import-Module](#) cmdlet to use an alternate prefix. The value of the [Prefix](#) parameter takes precedence over the value of the [DefaultCommandPrefix](#) key.

Supporting paths on non-Windows systems

Non-Windows platforms use the colon (:) character as a path separator and a forward-slash (/) character as a directory separator. The `[System.IO.Path]` class has static members that can be used to make your code work on any platform:

- `[System.IO.Path]::PathSeparator` - returns the character used to separate paths in a PATH environment variable for the host platform
- `[System.IO.Path]::DirectorySeparatorChar` - returns the character used to separate directory names with a path for the host platform

Use these static properties to in place of the ; and \ characters when you are constructing path strings.

See Also

[about_Command_Precedence](#)

[Writing a Windows PowerShell Module](#)

Registering Cmdlets

Article • 09/17/2021

The topics in this section describe modules and snap-ins and how to use modules and snap-ins to make cmdlets available in a Windows PowerShell session.

In This Section

[Modules and Snap-ins](#) Describes the differences between registering cmdlets through modules and through snap-ins.

[How to Register Cmdlets using Modules](#) Describes how to register cmdlets using modules.

[How to Create a Windows PowerShell Snap-in](#) Describes how to register cmdlets using snap-ins.

See Also

[Writing a Windows PowerShell Cmdlet](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Modules and Snap-ins

Article • 09/17/2021

Cmdlets can be added to a session using modules (introduced by Windows PowerShell 2.0) or snap-ins. Once the cmdlet is added to the session it can be run programmatically by a host application or interactively at the command line.

We recommend that you use modules as the delivery method for adding cmdlets to a session for the following reasons:

- Modules allow you to add cmdlets by loading the assembly where the cmdlet is defined. There is no need to implement a snap-in class.
- Modules allow you to add other resources, such as variables, functions, scripts, types and formatting files, and more.
- Snap-ins can be used only to add cmdlets and providers to the session.

See Also

[Writing a Windows PowerShell Module](#)

[Writing a Windows PowerShell Cmdlet](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

How to Import Cmdlets Using Modules

Article • 09/17/2021

This article describes how to import cmdlets to a PowerShell session by using a binary module.

ⓘ Note

The members of modules can include cmdlets, providers, functions, variables, aliases, and much more. Snap-ins can contain only cmdlets and providers.

How to load cmdlets using a module

1. Create a module folder that has the same name as the assembly file in which the cmdlets are implemented. In this procedure, the module folder is created in the Windows `system32` folder.

```
%SystemRoot%\system32\WindowsPowerShell\v1.0\Modules\mymodule
```

2. Make sure that the `PSModulePath` environment variable includes the path to your new module folder. By default, the system folder is already added to the `PSModulePath` environment variable. To view the `PSModulePath`, type:
`$Env:PSModulePath`.

3. Copy the cmdlet assembly into the module folder.
4. Add a module manifest file (`.psd1`) in the module's root folder. PowerShell uses the module manifest to import your module. For more information, see [How to Write a PowerShell Module Manifest](#).
5. Run the following command to add the cmdlets to the session:

```
Import-Module [Module_Name]
```

This procedure can be used to test your cmdlets. It adds all the cmdlets in the assembly to the session. For more information about modules, see [Writing a Windows PowerShell Module](#).

See also

How to Write a PowerShell Module Manifest

[Importing a PowerShell Module](#)

[Import-Module](#)

[Installing Modules](#)

[about_PSMODULEPATH](#)

[Writing a Windows PowerShell Cmdlet](#)

How to Create a Windows PowerShell Snap-in

Article • 03/24/2025

A Windows PowerShell snap-in provides a mechanism for registering sets of cmdlets and another Windows PowerShell provider with the shell, thus extending the functionality of the shell. A Windows PowerShell snap-in can register all the cmdlets and providers in a single assembly, or it can register a specific list of cmdlets and providers.

Snap-in assemblies should be installed in a protected directory, just as they would be with other operating systems. Otherwise, malicious users can replace an assembly with unsafe code.

Windows PowerShell Snap-in Classes

All Windows PowerShell snap-in classes derive from the [System.Management.Automation.PSSnapIn](#) or [System.Management.Automation.CustomPSSnapIn](#) classes.

Examples

[Writing a Windows PowerShell Snap-in](#): This example shows how to create a snap-in that is used to register all the cmdlets and providers in an assembly.

[Writing a Custom Windows PowerShell Snap-in](#): This example shows how to create a custom snap-in that is used to register a specific set of cmdlets and providers that might or might not exist in a single assembly.

See Also

[System.Management.Automation.PSSnapIn](#)

[System.Management.Automation.CustomPSSnapIn](#)

[Registering Cmdlets](#)

[Windows PowerShell Shell SDK](#)

Writing a Windows PowerShell Snap-in

Article • 09/17/2021

This example shows how to write a Windows PowerShell snap-in that can be used to register all the cmdlets and Windows PowerShell providers in an assembly.

With this type of snap-in, you do not select which cmdlets and providers you want to register. To write a snap-in that allows you to select what is registered, see [Writing a Custom Windows PowerShell Snap-in](#).

Writing a Windows PowerShell Snap-in

1. Add the RunInstallerAttribute attribute.
2. Create a public class that derives from the [System.Management.Automation.PSSnapIn](#) class.

In this example, the class name is "GetProcPSSnapIn01".

3. Add a public property for the name of the snap-in (required). When naming snap-ins, do not use any of the following characters: #, ., , (,), {, }, [,], &, -, /, \, \$, ;, :, ", ', <, >, |, ?, @, ^, *

In this example, the name of the snap-in is "GetProcPSSnapIn01".

4. Add a public property for the vendor of the snap-in (required).

In this example, the vendor is "Microsoft".

5. Add a public property for the vendor resource of the snap-in (optional).

In this example, the vendor resource is "GetProcPSSnapIn01,Microsoft".

6. Add a public property for the description of the snap-in (required).

In this example, the description is "This is a Windows PowerShell snap-in that registers the Get-Proc cmdlet".

7. Add a public property for the description resource of the snap-in (optional).

In this example, the vendor resource is "GetProcPSSnapIn01,This is a Windows PowerShell snap-in that registers the Get-Proc cmdlet".

Example

This example shows how to write a Windows PowerShell snap-in that can be used to register the Get-Proc cmdlet in the Windows PowerShell shell. Be aware that in this example, the complete assembly would contain only the GetProcPSSnapIn01 snap-in class and the `Get-Proc` cmdlet class.

C#

```
[RunInstaller(true)]
public class GetProcPSSnapIn01 : PSSnapIn
{
    /// <summary>
    /// Create an instance of the GetProcPSSnapIn01 class.
    /// </summary>
    public GetProcPSSnapIn01()
        : base()
    {

    }

    /// <summary>
    /// Specify the name of the PowerShell snap-in.
    /// </summary>
    public override string Name
    {
        get
        {
            return "GetProcPSSnapIn01";
        }
    }

    /// <summary>
    /// Specify the vendor for the PowerShell snap-in.
    /// </summary>
    public override string Vendor
    {
        get
        {
            return "Microsoft";
        }
    }

    /// <summary>
    /// Specify the localization resource information for the vendor.
    /// Use the format: resourceBaseName, VendorName.
    /// </summary>
    public override string VendorResource
    {
        get
        {
            return "GetProcPSSnapIn01, Microsoft";
        }
    }
}
```

```
}

/// <summary>
/// Specify a description of the PowerShell snap-in.
/// </summary>
public override string Description
{
    get
    {
        return "This is a PowerShell snap-in that includes the Get-Proc
cmdlet.";
    }
}

/// <summary>
/// Specify the localization resource information for the description.
/// Use the format: resourceBaseName,Description.
/// </summary>
public override string DescriptionResource
{
    get
    {
        return "GetProcPSSnapIn01,This is a PowerShell snap-in that includes
the Get-Proc cmdlet.";
    }
}
```

See Also

[How to Register Cmdlets, Providers, and Host Applications](#)

[Windows PowerShell Shell SDK](#)

Writing a Custom Windows PowerShell Snap-in

Article • 09/17/2021

This example shows how to write a Windows PowerShell snap-in that registers specific cmdlets.

With this type of snap-in, you specify which cmdlets, providers, types, or formats to register. For more information about how to write a snap-in that registers all the cmdlets and providers in an assembly, see [Writing a Windows PowerShell Snap-in](#).

To write a Windows PowerShell Snap-in that registers specific cmdlets.

1. Add the RunInstallerAttribute attribute.
2. Create a public class that derives from the [System.Management.Automation.CustomPSSnapIn](#) class.

In this example, the class name is "CustomPSSnapinTest".

3. Add a public property for the name of the snap-in (required). When naming snap-ins, do not use any of the following characters: #, ., , (,), {, }, [,], &, -, /, \, \$, ;, :, ", ', <, >, |, ?, @, `*, *

In this example, the name of the snap-in is "CustomPSSnapInTest".

4. Add a public property for the vendor of the snap-in (required).

In this example, the vendor is "Microsoft".

5. Add a public property for the vendor resource of the snap-in (optional).

In this example, the vendor resource is "CustomPSSnapInTest,Microsoft".

6. Add a public property for the description of the snap-in (required).

In this example, the description is: "This is a custom Windows PowerShell snap-in that includes the `Test-Helloworld` and `Test-CustomSnapinTest` cmdlets".

7. Add a public property for the description resource of the snap-in (optional).

In this example, the vendor resource is:

CustomPSSnapInTest, This is a custom Windows PowerShell snap-in that includes the Test-HelloWorld and Test-CustomSnapinTest cmdlets".

8. Specify the cmdlets that belong to the custom snap-in (optional) using the [System.Management.Automation.Runspaces.CmdletConfigurationEntry](#) class. The information added here includes the name of the cmdlet, its .NET type, and the cmdlet Help file name (the format of the cmdlet Help file name should be `name.dll-help.xml`).

This example adds the Test-HelloWorld and TestCustomSnapinTest cmdlets.

9. Specify the providers that belong to the custom snap-in (optional).

This example does not specify any providers.

10. Specify the types that belong to the custom snap-in (optional).

This example does not specify any types.

11. Specify the formats that belong to the custom snap-in (optional).

This example does not specify any formats.

Example

This example shows how to write a Custom Windows PowerShell snap-in that can be used to register the `Test-HelloWorld` and `Test-CustomSnapinTest` cmdlets. Be aware that in this example, the complete assembly could contain other cmdlets and providers that would not be registered by this snap-in.

C#

```
[RunInstaller(true)]
public class CustomPSSnapinTest : CustomPSSnapIn
{
    /// <summary>
    /// Creates an instance of CustomPSSnapInTest class.
    /// </summary>
    public CustomPSSnapinTest()
        : base()
    {

    }

    /// <summary>
    /// Specify the name of the custom PowerShell snap-in.
    /// </summary>
    public override string Name
    {
```

```
get
{
    return "CustomPSSnapInTest";
}
}

/// <summary>
/// Specify the vendor for the custom PowerShell snap-in.
/// </summary>
public override string Vendor
{
    get
    {
        return "Microsoft";
    }
}

/// <summary>
/// Specify the localization resource information for the vendor.
/// Use the format: resourceName,resourceName.
/// </summary>
public override string VendorResource
{
    get
    {
        return "CustomPSSnapInTest,Microsoft";
    }
}

/// <summary>
/// Specify a description of the custom PowerShell snap-in.
/// </summary>
public override string Description
{
    get
    {
        return "This is a custom PowerShell snap-in that includes the Test-
HelloWorld and Test-CustomSnapinTest cmdlets.";
    }
}

/// <summary>
/// Specify the localization resource information for the description.
/// Use the format: resourceName,Description.
/// </summary>
public override string DescriptionResource
{
    get
    {
        return "CustomPSSnapInTest,This is a custom PowerShell snap-in that
includes the Test-Helloworld and Test-CustomSnapinTest cmdlets.";
    }
}

/// <summary>
```

```
/// Specify the cmdlets that belong to this custom PowerShell snap-in.
/// </summary>
private Collection<CmdletConfigurationEntry> _cmdlets;
public override Collection<CmdletConfigurationEntry> Cmdlets
{
    get
    {
        if (_cmdlets == null)
        {
            _cmdlets = new Collection<CmdletConfigurationEntry>();
            _cmdlets.Add(new CmdletConfigurationEntry("test-customsnapintest",
typeof(TestCustomSnapinTest), "TestCmdletHelp.dll-help.xml"));
            _cmdlets.Add(new CmdletConfigurationEntry("test-helloworld",
typeof(TestHelloWorld), "HelloWorldHelp.dll-help.xml"));
        }

        return _cmdlets;
    }
}

/// <summary>
/// Specify the providers that belong to this custom PowerShell snap-in.
/// </summary>
private Collection<ProviderConfigurationEntry> _providers;
public override Collection<ProviderConfigurationEntry> Providers
{
    get
    {
        if (_providers == null)
        {
            _providers = new Collection<ProviderConfigurationEntry>();
        }

        return _providers;
    }
}

/// <summary>
/// Specify the types that belong to this custom PowerShell snap-in.
/// </summary>
private Collection<TypeConfigurationEntry> _types;
public override Collection<TypeConfigurationEntry> Types
{
    get
    {
        if (_types == null)
        {
            _types = new Collection<TypeConfigurationEntry>();
        }

        return _types;
    }
}

/// <summary>
```

```
/// Specify the formats that belong to this custom PowerShell snap-in.  
/// </summary>  
private Collection<FormatConfigurationEntry> _formats;  
public override Collection<FormatConfigurationEntry> Formats  
{  
    get  
    {  
        if (_formats == null)  
        {  
            _formats = new Collection<FormatConfigurationEntry>();  
        }  
  
        return _formats;  
    }  
}  
}
```

For more information about registering snap-ins, see [How to Register Cmdlets, Providers, and Host Applications](#) in the [Windows PowerShell Programmer's Guide](#).

See Also

[How to Register Cmdlets, Providers, and Host Applications](#)

[Windows PowerShell Shell SDK](#)

Importing a PowerShell Module

Article • 09/17/2021

Once you have installed a module on a system, you will likely want to import the module. Importing is the process that loads the module into active memory, so that a user can access that module in their PowerShell session. In PowerShell 2.0, you can import a newly-installed PowerShell module with a call to [Import-Module](#) cmdlet. In PowerShell 3.0, PowerShell is able to implicitly import a module when one of the functions or cmdlets in the module is called by a user. Note that both versions assume that you install your module in a location where PowerShell is able to find it; for more information, see [Installing a PowerShell Module](#). You can use a module manifest to restrict what parts of your module are exported, and you can use parameters of the `Import-Module` call to restrict what parts are imported.

Importing a Snap-In (PowerShell 1.0)

Modules did not exist in PowerShell 1.0: instead, you had to register and use snap-ins. However, it is not recommended that you use this technology at this point, as modules are generally easier to install and import. For more information, see [How to Create a Windows PowerShell Snap-in](#).

Importing a Module with Import-Module (PowerShell 2.0)

PowerShell 2.0 uses the appropriately-named [Import-Module](#) cmdlet to import modules. When this cmdlet is run, Windows PowerShell searches for the specified module within the directories specified in the `PSModulePath` variable. When the specified directory is found, Windows PowerShell searches for files in the following order: module manifest files (`.psd1`), script module files (`.psm1`), binary module files (`.dll`). For more information about adding directories to the search, see [about_PSModulePath](#). The following code describes how to import a module:

```
PowerShell
Import-Module myModule
```

Assuming that `myModule` was located in the `PSModulePath`, PowerShell would load `myModule` into active memory. If `myModule` was not located on a `PSModulePath` path,

you could still explicitly tell PowerShell where to find it:

```
PowerShell
```

```
Import-Module -Name C:\myRandomDirectory\myModule -Verbose
```

You can also use the `-Verbose` parameter to identify what is being exported out of the module, and what is being imported into active memory. Both exports and imports restrict what is exposed to the user: the difference is who is controlling the visibility. Essentially, exports are controlled by code within the module. In contrast, imports are controlled by the `Import-Module` call. For more information, see [Restricting Members That Are Imported](#), below.

Implicitly Importing a Module (PowerShell 3.0)

Beginning in Windows PowerShell 3.0, modules are imported automatically when any cmdlet or function in the module is used in a command. This feature works on any module in a directory that is included in the value of the **PSModulePath** environment variable. If you do not save your module on a valid path however, you can still load them using the explicit `Import-Module` option, described above.

The following actions trigger automatic importing of a module, also known as "module auto-loading."

- Using a cmdlet in a command. For example, typing `Get-ExecutionPolicy` imports the Microsoft.PowerShell.Security module that contains the `Get-ExecutionPolicy` cmdlet.
- Using the `Get-Command` cmdlet to get the command. For example, typing `Get-Command Get-JobTrigger` imports the **PSScheduledJob** module that contains the `Get-JobTrigger` cmdlet. A `Get-Command` command that includes wildcard characters is considered to be discovery and does not trigger importing of a module.
- Using the `Get-Help` cmdlet to get help for a cmdlet. For example, typing `Get-Help Get-WinEvent` imports the Microsoft.PowerShell.Diagnostics module that contains the `Get-WinEvent` cmdlet.

To support automatic importing of modules, the `Get-Command` cmdlet gets all cmdlets and functions in all installed modules, even if the module is not imported into the session. For more information, see the help topic for the `Get-Command` cmdlet.

The Importing Process

When a module is imported, a new session state is created for the module, and a `System.Management.Automation.PSModuleInfo` object is created in memory. A session-state is created for each module that is imported (this includes the root module and any nested modules). The members that are exported from the root module, including any members that were exported to the root module by any nested modules, are then imported into the caller's session state.

The metadata of members that are exported from a module have a `ModuleName` property. This property is populated with the name of the module that exported them.

⚠️ Warning

If the name of an exported member uses an unapproved verb or if the name of the member uses restricted characters, a warning is displayed when the [Import-Module](#) cmdlet is run.

By default, the [Import-Module](#) cmdlet does not return any objects to the pipeline. However, the cmdlet supports a `PassThru` parameter that can be used to return a `System.Management.Automation.PSModuleInfo` object for each module that is imported. To send output to the host, users should run the [Write-Host](#) cmdlet.

Restricting the Members That Are Imported

When a module is imported by using the [Import-Module](#) cmdlet, by default, all exported module members are imported into the session, including any commands exported to the module by a nested module. By default, variables and aliases are not exported. To restrict the members that are exported, use a [module manifest](#). To restrict the members that are imported, use the following parameters of the `Import-Module` cmdlet.

- **Function:** This parameter restricts the functions that are exported. (If you are using a module manifest, see the `FunctionsToExport` key.)
- **`Cmdlet`:** This parameter restricts the cmdlets that are exported (If you are using a module manifest, see the `CmdletsToExport` key.)
- **Variable:** This parameter restricts the variables that are exported (If you are using a module manifest, see the `VariablesToExport` key.)
- **Alias:** This parameter restricts the aliases that are exported (If you are using a module manifest, see the `AliasesToExport` key.)

See Also

[Writing a Windows PowerShell Module](#)

Windows PowerShell Provider Quickstart

Article • 03/24/2025

This topic explains how to create a Windows PowerShell provider that has basic functionality of creating a new drive. For general information about providers, see [Windows PowerShell Provider Overview](#). For examples of providers with more complete functionality, see [Provider Samples](#).

Writing a basic provider

The most basic functionality of a Windows PowerShell provider is to create and remove drives. In this example, we implement the

`System.Management.Automation.Provider.DriveCmdletProvider.NewDrive*` and `System.Management.Automation.Provider.DriveCmdletProvider.RemoveDrive*` methods of the `System.Management.Automation.Provider.DriveCmdletProvider` class. You will also see how to declare a provider class.

When you write a provider, you can specify default drives—drives that are created automatically when the provider is available. You also define a method to create new drives that use that provider.

The examples provided in this topic are based on the [AccessDBProviderSample02](#) sample, which is part of a larger sample that represents an Access database as a Windows PowerShell drive.

Setting up the project

In Visual Studio, create a Class Library project named `AccessDBProviderSample`. Complete the following steps to configure your project so that Windows PowerShell will start, and the provider will be loaded into the session, when you build and start your project.

Configure the provider project

1. Add the `System.Management.Automation` assembly as a reference to your project.
2. Click **Project > AccessDBProviderSample Properties > Debug**. In **Start project**, click **Start external program**, and navigate to the Windows PowerShell executable (typically `C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe`).

3. Under **Start Options**, enter the following into the **Command line arguments** box:

```
-NoExit -Command "[Reflection.Assembly]::LoadFrom(AccessDBProviderSample.dll') | Import-Module"
```

Declaring the provider class

Our provider derives from the

[System.Management.Automation.Provider.DriveCmdletProvider](#) class. Most providers that provide real functionality (accessing and manipulating items, navigating the data store, and getting and setting content of items) derive from the

[System.Management.Automation.Provider.NavigationCmdletProvider](#) class.

In addition to specifying that the class derives from

[System.Management.Automation.Provider.DriveCmdletProvider](#), you must decorate it with the [System.Management.Automation.Provider.CmdletProviderAttribute](#) as shown in the example.

C#

```
namespace Microsoft.Samples.PowerShell.Providers
{
    using System;
    using System.Data;
    using System.Data.Odbc;
    using System.IO;
    using System.Management.Automation;
    using System.Management.Automation.Provider;

    #region AccessDBProvider

    [CmdletProvider("AccessDB", ProviderCapabilities.None)]
    public class AccessDBProvider : DriveCmdletProvider
    {

    }
}
```

Implementing NewDrive

The [System.Management.Automation.Provider.DriveCmdletProvider.NewDrive*](#) method is called by the Windows PowerShell engine when a user calls the [Microsoft.PowerShell.Commands.NewPSDriveCommand](#) cmdlet specifying the name of your provider. The PSDriveInfo parameter is passed by the Windows PowerShell engine, and the method returns the new drive to the Windows PowerShell engine. This method must be declared within the class created above.

The method first checks to make sure both the drive object and the drive root that were passed in exist, returning `null` if either of them do not. It then uses a constructor of the internal class `AccessDBPSDriveInfo` to create a new drive and a connection to the Access database the drive represents.

C#

```
protected override PSDriveInfo NewDrive(PSDriveInfo drive)
{
    // Check if the drive object is null.
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            null));

        return null;
    }

    // Check if the drive root is not null or empty
    // and if it is an existing file.
    if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root) ==
false))
    {
        WriteError(new ErrorRecord(
            new ArgumentException("drive.Root"),
            "NoRoot",
            ErrorCategory.InvalidArgument,
            drive));

        return null;
    }

    // Create a new drive and create an ODBC connection to the new drive.
    AccessDBPSDriveInfo accessDBPSDriveInfo = new
AccessDBPSDriveInfo(drive);
    OdbcConnectionStringBuilder builder = new
OdbcConnectionStringBuilder();

    builder.Driver = "Microsoft Access Driver (*.mdb)";
    builder.Add("DBQ", drive.Root);

    OdbcConnection conn = new OdbcConnection(builder.ConnectionString);
    conn.Open();
    accessDBPSDriveInfo.Connection = conn;

    return accessDBPSDriveInfo;
}
```

The following is the AccessDBPSDriveInfo internal class that includes the constructor used to create a new drive, and contains the state information for the drive.

```
C#  
  
internal class AccessDBPSDriveInfo : PSDriveInfo  
{  
    /// <summary>  
    /// A reference to the connection to the database.  
    /// </summary>  
    private OdbcConnection connection;  
  
    /// <summary>  
    /// Initializes a new instance of the AccessDBPSDriveInfo class.  
    /// The constructor takes a single argument.  
    /// </summary>  
    /// <param name="driveInfo">Drive defined by this provider</param>  
    public AccessDBPSDriveInfo(PSDriveInfo driveInfo)  
        : base(driveInfo)  
    {  
    }  
  
    /// <summary>  
    /// Gets or sets the ODBC connection information.  
    /// </summary>  
    public OdbcConnection Connection  
    {  
        get { return this.connection; }  
        set { this.connection = value; }  
    }  
}
```

Implementing RemoveDrive

The [System.Management.Automation.Provider.DriveCmdletProvider.RemoveDrive](#)* method is called by the Windows PowerShell engine when a user calls the [Microsoft.PowerShell.Commands.RemovePSDriveCommand](#) cmdlet. The method in this provider closes the connection to the Access database.

```
C#  
  
protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)  
{  
    // Check if drive object is null.  
    if (drive == null)  
    {  
        WriteError(new ErrorRecord(  
            new ArgumentNullException("drive"),  
            "NullDrive",  
            ErrorCategory.InvalidArgument,
```

```
        drive));  
  
    return null;  
}  
  
// Close the ODBC connection to the drive.  
AccessDBPSDriveInfo accessDBPSDriveInfo = drive as  
AccessDBPSDriveInfo;  
  
if (accessDBPSDriveInfo == null)  
{  
    return null;  
}  
  
accessDBPSDriveInfo.Connection.Close();  
  
return accessDBPSDriveInfo;  
}
```

Windows PowerShell Provider Overview

Article • 09/17/2021

A Windows PowerShell provider allows any data store to be exposed like a file system as if it were a mounted drive. For example, the built-in Registry provider allows you to navigate the registry like you would navigate the `c` drive of your computer. A provider can also override the `Item` cmdlets (for example, `Get-Item`, `Set-Item`, etc.) such that the data in your data store can be treated like files and directories are treated when navigating a file system. For more information about providers and drives, and the built-in providers in Windows PowerShell, see [about_Providers](#).

Providers and Drives

A Provider defines the logic that is used to access, navigate, and edit a data store, while a drive specifies a specific entry point to a data store (or a portion of a data store) that is of the type defined by the provider. For example, the Registry provider allows you to access hives and keys in a registry, and the HKLM and HKCU drives specify the corresponding hives within the registry. The HKLM and HKCU drives both use the Registry provider.

When you write a provider, you can specify default drives—drives that are created automatically when the provider is available. You also define a method to create new drives that use that provider.

Type of Providers

There are several types of providers, each of which provides a different level of functionality. A provider is implemented as a class that derives from one of the descendants of the [System.Management.Automation.SessionStateCategory CmdletProvider](#) class. For information about the different types of providers, see [Provider types](#).

Provider cmdlets

Providers can implement methods that correspond to cmdlets, creating custom behaviors for those cmdlets when used in a drive for that provider. Depending on the type of provider, different sets of cmdlets are available. For a complete list of the cmdlets available for customization in providers, see [Provider cmdlets](#).

Provider paths

Users navigate provider drives like file systems. Because of this, they expect the syntax of paths to correspond to the paths used in file system navigation. When a user runs a provider cmdlet, they specify a path to the item to be accessed. The path that is specified can be interpreted in several ways. A provider should support one or more of the following path types.

Drive-qualified paths

A drive-qualified path is a combination of the item name, the container and subcontainers in which the item is located, and the Windows PowerShell drive through which the item is accessed. (Drives are defined by the provider that is used to access the data store. This path starts with the drive name followed by a colon (:). For example:

```
Get-ChildItem C:
```

Provider-qualified paths

To allow the Windows PowerShell engine to initialize and uninitialized your provider, the provider must support a provider-qualified path. For example, the user can initialize and uninitialized the FileSystem provider because it defines the following provider-qualified path: `FileSystem::\\uncshare\abc\bar`.

Provider-direct paths

To allow remote access to your Windows PowerShell provider, it should support a provider-direct path to pass directly to the Windows PowerShell provider for the current location. For example, the registry Windows PowerShell provider can use `\\server\regkeypath` as a provider-direct path.

Provider-internal paths

To allow the provider cmdlet to access data using non-Windows PowerShell application programming interfaces (APIs), your Windows PowerShell provider should support a provider-internal path. This path is indicated after the ":" in the provider-qualified path. For example, the provider-internal path for the FileSystem Windows PowerShell provider is `\\uncshare\abc\bar`.

Overriding cmdlet parameters

The behavior of some provider-specific cmdlets can be overridden by a provider. For a list of parameters that can be overridden, and how to override them in your provider class, see [Provider cmdlet parameters](#)

Dynamic parameters

Providers can define dynamic parameters that are added to a provider cmdlet when the user specifies a certain value for one of the static parameters of the cmdlet. A provider does this by implementing one or more dynamic parameter methods. For a list of cmdlet parameters that can be used to add dynamic parameter, and the methods used to implement them, see [Provider cmdlet dynamic parameters](#).

Provider capabilities

The [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration defines a number of capabilities that providers can support. These include the ability to use wildcards, filter items, and support transactions. To specify capabilities for a provider, add a list of values of the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration, combined with a logical `OR` operation, as the [System.Management.Automation.Provider.CmdletProviderAttribute.ProviderCapabilities*](#) property (the second parameter of the attribute) of the [System.Management.Automation.Provider.CmdletProviderAttribute](#) attribute for your provider class. For example, the following attribute specifies that the provider supports the [System.Management.Automation.Provider.ProviderCapabilities ShouldProcess](#) and [System.Management.Automation.Provider.ProviderCapabilities Transactions](#) capabilities.

```
C#
```

```
[CmdletProvider(RegistryProvider.ProviderName,  
ProviderCapabilities.ShouldProcess | ProviderCapabilities.Transactions)]
```

Provider cmdlet help

When writing a provider, you can implement your own Help for the provider cmdlets that you support. This includes a single help topic for each provider cmdlet or multiple versions of a help topic for cases where the provider cmdlet acts differently based on the use of dynamic parameters. To support provider cmdlet-specific help, your provider must implement the [System.Management.Automation.Provider.ICmdletProviderSupportsHelp](#) interface.

The Windows PowerShell engine calls the [System.Management.Automation.Provider.ICmdletProviderSupportsHelp.GetHelpMaml](#)* method to display the Help topic for your provider cmdlets. The engine provides the name of the cmdlet that the user specified when running the `Get-Help` cmdlet and the current path of the user. The current path is required if your provider implements different versions of the same provider cmdlet for different drives. The method must return a string that contains the XML for the cmdlet Help.

The content for the Help file is written using PSMAML XML. This is the same XML schema that is used for writing Help content for stand-alone cmdlets. Add the content for your custom cmdlet Help to the Help file for your provider under the `CmdletHelpPaths` element. The following example shows the `command` element for a single provider cmdlet, and it shows how you specify the name of the provider cmdlet that your provider supports

XML

```
<CmdletHelpPaths>
  <command:command>
    <command:details>
      <command:name>ProviderCmdletName</command:name>
      <command:verb>Verb</command:verb>
      <command:noun>Noun</command:noun>
      <command:details>
    </command:command>
  <CmdletHelpPath>
```

See Also

[Windows PowerShell Provider Functionality](#)

[Provider Cmdlets](#)

[Writing a Windows PowerShell Provider](#)

Provider types

Article • 01/31/2024

Providers define their basic functionality by changing how the provider cmdlets, provided by PowerShell, perform their actions. For example, providers can use the default functionality of the `Get-Item` cmdlet, or they can change how that cmdlet operates when retrieving items from the data store. The provider functionality described in this document includes functionality defined by overwriting methods from specific provider base classes and interfaces.

ⓘ Note

For provider features that are pre-defined by PowerShell, see [Provider capabilities](#).

Drive-enabled providers

Drive-enabled providers specify the default drives available to the user and allow the user to add or remove drives. In most cases, providers are drive-enabled providers because they require some default drive to access the data store. However, when writing your own provider you might or might not want to allow the user to create and remove drives.

To create a drive-enabled provider, your provider class must derive from the [System.Management.Automation.Provider.DriveCmdletProvider](#) class or another class that derives from that class. The **DriveCmdletProvider** class defines the following methods for implementing the default drives of the provider and supporting the `New-PSDrive` and `Remove-PSDrive` cmdlets. In most cases, to support a provider cmdlet you must overwrite the method that the PowerShell engine calls to invoke the cmdlet, such as the `NewDrive` method for the `New-PSDrive` cmdlet, and optionally you can overwrite a second method, such as `NewDriveDynamicParameters`, for adding dynamic parameters to the cmdlet.

- The `InitializeDefaultDrives` method defines the default drives that are available to the user whenever the provider is used.
- The `NewDrive` and `NewDriveDynamicParameters` methods defines how your provider supports the `New-PSDrive` provider cmdlet. This cmdlet allows the user to create drives to access the data store.

- The [RemoveDrive](#) method defines how your provider supports the `Remove-PSDrive` provider cmdlet. This cmdlet allows the user to remove drives from the data store.

Item-enabled providers

Item-enabled providers allow the user to get, set, or clear the items in the data store. An "item" is an element of the data store that the user can access or manage independently. To create an item-enabled provider, your provider class must derive from the [System.Management.Automation.Provider.ItemCmdletProvider](#) class or another class that derives from that class.

The [ItemCmdletProvider](#) class defines the following methods for implementing specific provider cmdlets. In most cases, to support a provider cmdlet you must overwrite the method that the PowerShell engine calls to invoke the cmdlet, such as the `ClearItem` method for the `Clear-Item` cmdlet, and optionally you can overwrite a second method, such as `ClearItemDynamicParameters`, for adding dynamic parameters to the cmdlet.

- The [ClearItem](#) and [ClearItemDynamicParameters](#) methods define how your provider supports the `Clear-Item` provider cmdlet. This cmdlet allows the user to remove of the value of an item in the data store.
- The [GetItem](#) and [GetItemDynamicParameters](#) methods define how your provider supports the `Get-Item` provider cmdlet. This cmdlet allows the user to retrieve data from the data store.
- The [SetItem](#) and [SetItemDynamicParameters](#) methods define how your provider supports the `Set-Item` provider cmdlet. This cmdlet allows the user to update the values of items in the data store.
- The [InvokeDefaultAction](#) and [InvokeDefaultActionDynamicParameters](#) methods define how your provider supports the `Invoke-Item` provider cmdlet. This cmdlet allows the user to perform the default action specified by the item.
- The [ItemExists](#) and [ItemExistsDynamicParameters](#) methods define how your provider supports the `Test-Path` provider cmdlet. This cmdlet allows the user to determine if all the elements of a path exist.

In addition to the methods used to implement provider cmdlets, the [ItemCmdletProvider](#) class also defines the following methods:

- The [ExpandPath](#) method allows the user to use wildcards when specifying the provider path.

- The [IsValidPath](#) is used to determine if a path is syntactically and semantically valid for the provider.

Container-enabled providers

Container-enabled providers allow the user to manage items that are containers. A container is a group of child items under a common parent item. To create a container-enabled provider, your provider class must derive from the [System.Management.Automation.Provider.ContainerCmdletProvider](#) class or another class that derives from that class.

Important

Container-enabled providers can't access data stores that contain nested containers. If a child item of a container is another container, you must implement a navigation-enabled provider.

The [ContainerCmdletProvider](#) class defines the following methods for implementing specific provider cmdlets. In most cases, to support a provider cmdlet you must overwrite the method that the PowerShell engine calls to invoke the cmdlet, such as the `CopyItem` method for the `Copy-Item` cmdlet, and optionally you can overwrite a second method, such as `CopyItemDynamicParameters`, for adding dynamic parameters to the cmdlet.

- The [CopyItem](#) and [CopyItemDynamicParameters](#) methods define how your provider supports the `Copy-Item` provider cmdlet. This cmdlet allows the user to copy an item from one location to another.
- The [GetChildItems](#) and [GetChildItemsDynamicParameters](#) methods define how your provider supports the `Get-ChildItem` provider cmdlet. This cmdlet allows the user to retrieve the child items of the parent item.
- The [GetChildNames](#) and [GetChildNamesDynamicParameters](#) methods define how your provider supports the `Get-ChildItem` provider cmdlet if its `Name` parameter is specified.
- The [NewItem](#) and [NewItemDynamicParameters](#) methods define how your provider supports the `New-Item` provider cmdlet. This cmdlet allows the user to create new items in the data store.

- The [RemoveItem](#) and [RemoveItemDynamicParameters](#) methods define how your provider supports the `Remove-Item` provider cmdlet. This cmdlet allows the user to remove items from the data store.
- The [RenameItem](#) and [RenameItemDynamicParameters](#) methods define how your provider supports the `Rename-Item` provider cmdlet. This cmdlet allows the user to rename items in the data store.

In addition to the methods used to implement provider cmdlets, the [ContainerCmdletProvider](#) class also defines the following methods:

- The [HasChildItems](#) method can be used by the provider class to determine whether an item has child items.
- The [ConvertPath](#) method can be used by the provider class to create a new provider-specific path from a specified path.

Navigation-enabled providers

Navigation-enabled providers allow the user to move items in the data store. To create a navigation-enabled provider, your provider class must derive from the [System.Management.Automation.Provider.NavigationCmdletProvider](#) class.

The [NavigationCmdletProvider](#) class defines the following methods for implementing specific provider cmdlets. In most cases, to support a provider cmdlet you must overwrite the method that the PowerShell engine calls to invoke the cmdlet, such as the `MoveItem` method for the `Move-Item` cmdlet, and optionally you can overwrite a second method, such as `MoveItemDynamicParameters`, for adding dynamic parameters to the cmdlet.

- The [MoveItem](#) and [MoveItemDynamicParameters](#) methods define how your provider supports the `Move-Item` provider cmdlet. This cmdlet allows the user to move an item from one location in the store to another location.
- The [MakePath](#) method defines how your provider supports the `Join-Path` provider cmdlet. This cmdlet allows the user to combine a parent and child path segment to create a provider-internal path.

In addition to the methods used to implement provider cmdlets, the [NavigationCmdletProvider](#) class also defines the following methods:

- The [GetChildName](#) method extracts the name of the child node of a path.

- The [GetParentPath](#) method extracts the parent part of a path.
- The [IsItemContainer](#) method determines whether the item is a container item. In this context, a container is a group of child items under a common parent item.
- The [NormalizeRelativePath](#) method returns a path to an item that's relative to a specified base path.

Content-enabled providers

Content-enabled providers allow the user to clear, get, or set the content of items in a data store. For example, the [FileSystem provider](#) allows you to clear, get, and set the content of files in the file system. To create a content enabled provider, your provider class must implement the methods of the [System.Management.Automation.Provider.IContentCmdletProvider](#) interface.

The [IContentCmdletProvider](#) interface defines the following methods for implementing specific provider cmdlets. In most cases, to support a provider cmdlet you must overwrite the method that the PowerShell engine calls to invoke the cmdlet, such as the `ClearContent` method for the `Clear-Content` cmdlet, and optionally you can overwrite a second method, such as `ClearContentDynamicParameters`, for adding dynamic parameters to the cmdlet.

- The [ClearContent](#) and [ClearContentDynamicParameters](#) methods define how your provider supports the `Clear-Content` provider cmdlet. This cmdlet allows the user to delete the content of an item without deleting the item.
- The [GetContentReader](#) and [GetContentReaderDynamicParameters](#) methods define how your provider supports the `Get-Content` provider cmdlet. This cmdlet allows the user to retrieve the content of an item. The `GetContentReader` method returns an [System.Management.Automation.Provider.IContentReader](#) interface that defines the methods used to read the content.
- The [GetContentWriter](#) and [GetContentWriterDynamicParameters](#) methods define how your provider supports the `Set-Content` provider cmdlet. This cmdlet allows the user to update the content of an item. The `GetContentWriter` method returns an [System.Management.Automation.Provider.IContentWriter](#) interface that defines the methods used to write the content.

Property-enabled providers

Property-enabled providers allow the user to manage the properties of the items in the data store. To create a property-enabled provider, your provider class must implement the methods of the [System.Management.Automation.Provider.IPropertyCmdletProvider](#) and [System.Management.Automation.Provider.IDynamicPropertyCmdletProvider](#) interfaces. In most cases, to support a provider cmdlet you must overwrite the method that the PowerShell engine calls to invoke the cmdlet, such as the `ClearProperty` method for the Clear-Property cmdlet, and optionally you can overwrite a second method, such as `ClearPropertyDynamicParameters`, for adding dynamic parameters to the cmdlet.

The **IPropertyCmdletProvider** interface defines the following methods for implementing specific provider cmdlets:

- The [ClearProperty](#) and [ClearPropertyDynamicParameters](#) methods define how your provider supports the `Clear-ItemProperty` provider cmdlet. This cmdlet allows the user to delete the value of a property.
- The [GetProperty](#) and [GetPropertyDynamicParameters](#) methods define how your provider supports the `Get-ItemProperty` provider cmdlet. This cmdlet allows the user to retrieve the property of an item.
- The [SetProperty](#) and [SetPropertyDynamicParameters](#) methods define how your provider supports the `Set-ItemProperty` provider cmdlet. This cmdlet allows the user to update the properties of an item.

The **IDynamicPropertyCmdletProvider** interface defines the following methods for implementing specific provider cmdlets:

- The [CopyProperty](#) and [CopyPropertyDynamicParameters](#) methods define how your provider supports the `Copy-ItemProperty` provider cmdlet. This cmdlet allows the user to copy a property and its value from one location to another.
- The [MoveProperty](#) and [MovePropertyDynamicParameters](#) methods define how your provider supports the `Move-ItemProperty` provider cmdlet. This cmdlet allows the user to move a property and its value from one location to another.
- The [NewProperty](#) and [NewPropertyDynamicParameters](#) methods define how your provider supports the `New-ItemProperty` provider cmdlet. This cmdlet allows the user to create a new property and set its value.
- The [RemoveProperty](#) and [RemovePropertyDynamicParameters](#) methods define how your provider supports the `Remove-ItemProperty` cmdlet. This cmdlet allows the user to delete a property and its value.

- The [RenameProperty](#) and [RenamePropertyDynamicParameters](#) methods define how your provider supports the `Rename-ItemProperty` cmdlet. This cmdlet allows the user to change the name of a property.

See also

[about_Providers](#)

[Writing a Windows PowerShell Provider](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Provider cmdlets

Article • 01/31/2024

The cmdlets that the user can run to manage a data store are referred to as provider cmdlets. To support these cmdlets, you need to overwrite some of the methods defined by the base provider classes and interfaces.

Here are the provider cmdlets that can be run by the user:

PSDrive cmdlets

Get-PSDrive

This cmdlet returns the PowerShell drives in the current session. You do not need to overwrite any methods to support this cmdlet.

New-PSDrive

This cmdlet allows the user to create PowerShell drives to access the data store. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.DriveCmdletProvider](#) class:

- [NewDrive](#)
- [NewDriveDynamicParameters](#)

Remove-PSDrive

This cmdlet allows the user to remove PowerShell drives that access the data store. To support this cmdlet, overwrite the [System.Management.Automation.Provider.DriveCmdletProvider.RemoveDrive](#) method.

Item cmdlets

Clear-Item

This cmdlet allows the user to remove the value of an item in the data store. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.ItemCmdletProvider](#) class:

- [ClearItem](#)
- [ClearItemDynamicParameters](#)

Copy-Item

This cmdlet allows the user to copy an item from one location to another. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.ContainerCmdletProvider](#) class:

- [CopyItem](#)
- [CopyItemDynamicParameters](#)

Get-Item

This cmdlet allows the user to retrieve data from the data store. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.ItemCmdletProvider](#) class:

- [GetItem](#)
- [GetItemDynamicParameters](#)

Get-ChildItem

This cmdlet allows the user to retrieve the child items of the parent item. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.ContainerCmdletProvider](#) class:

- [GetChildItems](#)
- [GetChildItemsDynamicParameters](#)
- [GetChildNames](#)
- [GetChildNamesDynamicParameters](#)

Invoke-Item

This cmdlet allows the user to perform the default action specified by the item. To support this cmdlet, overwrite the [System.Management.Automation.Provider.ItemCmdletProvider.InvokeDefaultAction](#) method.

Move-Item

This cmdlet allows the user to move an item from one location to another location. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.NavigationCmdletProvider](#) class:

- [MoveItem](#)
- [MoveItemDynamicParameters](#)

New-ItemProperty

This cmdlet allows the user to create a new item in the data store.

Remove-Item

This cmdlet allows the user to remove items from the data store. To support this cmdlet, overwrite the following methods of

[System.Management.Automation.Provider.ContainerCmdletProvider](#) class:

- [RemoveItem](#)
- [RemoveItemDynamicParameters](#)

Rename-Item

This cmdlet allows the user to rename items in the data store. To support this cmdlet, overwrite the following methods of

[System.Management.Automation.Provider.ContainerCmdletProvider](#) class:

- [RenameItem](#)
- [RenameItemDynamicParameters](#)

Set-Item

This cmdlet allows the user to update the values of items in the data store. To support this cmdlet, overwrite the following methods of

[System.Management.Automation.Provider.ItemCmdletProvider](#) class:

- [SetItem](#)
- [SetItemDynamicParameters](#)

Item content cmdlets

Add-Content

This cmdlet allows the user to add content to an item.

Clear-Content

This cmdlet allows the user to delete content from an item without deleting the item. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.IContentCmdletProvider](#) interface:

- [ClearContent](#)
- [ClearContentDynamicParameters](#)

Get-Content

This cmdlet allows the user to retrieve the content of an item. To support this cmdlet, overwrite the following methods of

[System.Management.Automation.Provider.IContentCmdletProvider](#) interface:

- [GetContentReader](#)
- [GetContentReaderDynamicParameters](#)

The [GetContentReader](#) method returns an

[System.Management.Automation.Provider.IContentReader](#) interface that defines the methods used to read the content.

Set-Content

This cmdlet allows the user to update the content of an item. To support this cmdlet, overwrite the following methods of

[System.Management.Automation.Provider.IContentCmdletProvider](#) interface:

- [GetContentWriter](#)
- [GetContentWriterDynamicParameters](#)

The [GetContentWriter](#) method returns an

[System.Management.Automation.Provider.IContentWriter](#) interface that defines the methods used to write the content.

Item property cmdlets

Clear-ItemProperty

This cmdlet allows the user to delete the value of a property. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.IPropertyCmdletProvider](#) interface:

- [ClearProperty](#)
- [ClearPropertyDynamicParameters](#)

Copy-ItemProperty

This cmdlet allows the user to copy a property and its value from one location to another. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.IDynamicPropertyCmdletProvider](#) interface:

- [CopyProperty](#)
- [CopyPropertyDynamicParameters](#)

Get-ItemProperty

This cmdlet retrieves the properties of an item. To support this cmdlet, overwrite the following methods of

[System.Management.Automation.Provider.IPropertyCmdletProvider](#) interface:

- [GetProperty](#)
- [GetPropertyDynamicParameters](#)

Move-ItemProperty

This cmdlet allows the user to move a property and its value from one location to another. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.IDynamicPropertyCmdletProvider](#) interface:

- [MoveProperty](#)
- [MovePropertyDynamicParameters](#)

New-ItemProperty

This cmdlet allows the user to create a new property and set its value. To support this cmdlet, overwrite the following methods of

[System.Management.Automation.Provider.IDynamicPropertyCmdletProvider](#) interface:

- [NewProperty](#)
- [NewPropertyDynamicParameters](#)

Remove-ItemProperty

This cmdlet allows the user to delete a property and its value. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.IDynamicPropertyCmdletProvider](#) interface:

- [RemoveProperty](#)
- [RemovePropertyDynamicParameters](#)

Rename-ItemProperty

This cmdlet allows the user to change the name of a property. To support this cmdlet, overwrite the following methods of

[System.Management.Automation.Provider.IDynamicPropertyCmdletProvider](#) interface:

- [RenameProperty](#)
- [RenamePropertyDynamicParameters](#)

Set-ItemProperty

This cmdlet allows the user to update the properties of an item. To support this cmdlet, overwrite the following methods of

[System.Management.Automation.Provider.IPropertyCmdletProvider](#) interface:

- [SetProperty](#)
- [SetPropertyDynamicParameters](#)

Location cmdlets

Get-Location

Retrieves information about the current working location. You do not need to overwrite any methods to support this cmdlet.

Pop-Location

This cmdlet changes the current location to the location most recently pushed onto the stack. You do not need to overwrite any methods to support this cmdlet.

Push-Location

This cmdlet adds the current location to the top of a list of locations (a "stack"). You do not need to overwrite any methods to support this cmdlet.

Set-Location

This cmdlet sets the current working location to a specified location. You do not need to overwrite any methods to support this cmdlet.

Path cmdlets

Join-Path

This cmdlet allows the user to combine a parent and child path segment to create a provider-internal path. To support this cmdlet, overwrite the [System.Management.Automation.Provider.NavigationCmdletProvider.MakePath](#) method.

Convert-Path

This cmdlet converts a path from a PowerShell path to a PowerShell provider path.

Split-Path

Returns the specified part of a path.

Resolve-Path

Resolves the wildcard characters in a path, and displays the path contents.

Test-Path

This cmdlet determines whether all elements of a path exist. To support this cmdlet, overwrite the following methods of [System.Management.Automation.Provider.ItemCmdletProvider](#) class:

- [ItemExists](#)
- [ItemExistsDynamicParameters](#)

PSProvider cmdlets

Get-PSProvider

This cmdlet returns information about the providers available in the session. You do not need to overwrite any methods to support this cmdlet.

Provider cmdlet parameters

Article • 03/24/2025

Provider cmdlets come with a set of static parameters that are available to all providers that support the cmdlet, as well as dynamic parameters that are added when the user specifies a certain value for certain static parameters of the provider cmdlet.

Provider Cmdlet Static Parameters

Static parameters are defined by Windows PowerShell. A large set of these parameters is implemented by Windows PowerShell to provide consistency across all the providers and to provide a simpler development experience. Examples of these parameters include the `LiteralPath`, `Exclude`, and `Include` parameters of the `Get-Item` cmdlet. A smaller set of these parameters can be overwritten to provide actions that are specific to your provider. Examples of these parameters include the `Path` and `Value` parameter of the `Set-Item` cmdlet. Here is a list of the parameters that can be overwritten for the provider cmdlets.

`Clear-Content` cmdlet You can define how your provider will use the values passed to the `Path` parameter of the `Clear-Content` cmdlet by implementing the [System.Management.Automation.Provider.IContentCmdletProvider.ClearContent*](#) method.

`Clear-Item` cmdlet You can define how your provider will use the values passed to the `Path` parameter of the `Clear-Item` cmdlet by implementing the [System.Management.Automation.Provider.ItemCmdletProvider.ClearItem*](#) method.

`Clear-ItemProperty` cmdlet You can define how your provider will use the values passed to the `Path` and `Name` parameters of the `Clear-ItemProperty` cmdlet by implementing the [System.Management.Automation.Provider.IPropertyCmdletProvider.ClearProperty*](#) method.

`Copy-Item` cmdlet You can define how your provider will use the values passed to the `Path`, `Destination`, and `Recurse` parameters of the `Copy-Item` cmdlet by implementing the [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItem](#) method.

`Get-ChildItems` cmdlet You can define how your provider will use the values passed to the `Path` and `Recurse` parameters of the `Get-ChildItem` cmdlet by implementing the [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems*](#) and

`System.Management.Automation.Provider.ContainerCmdletProvider.GetChildNames*` methods.

`Get-Content` cmdlet You can define how your provider will use the values passed to the `Path` parameter of the `Get-Content` cmdlet by implementing the `System.Management.Automation.Provider.IContentCmdletProvider.GetContentReader*` method.

`Get-Item` cmdlet You can define how your provider will use the values passed to the `Path` parameter of the `Get-Item` cmdlet by implementing the `System.Management.Automation.Provider.ItemCmdletProvider.GetItem*` method.

`Get-ItemProperty` cmdlet You can define how your provider will use the values passed to the `Path` and `Name` parameters of the `Get-ItemProperty` cmdlet by implementing the `System.Management.Automation.Provider.IPropertyCmdletProvider.GetProperty*` method.

`Invoke-Item` cmdlet You can define how your provider will use the values passed to the `Path` parameter of the `Invoke-Item` cmdlet by implementing the `System.Management.Automation.Provider.ItemCmdletProvider.InvokeDefaultAction*` method.

`Move-Item` cmdlet You can define how your provider will use the values passed to the `Path` and `Destination` parameters of the `Move-Item` cmdlet by implementing the `System.Management.Automation.Provider.NavigationCmdletProvider.MoveItem*` method.

`New-Item` cmdlet You can define how your provider will use the values passed to the `Path`, `ItemType`, and `Value` parameters of the `New-Item` cmdlet by implementing the `System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*` method.

`New-ItemProperty` cmdlet You can define how your provider will use the values passed to the `Path`, `Name`, `.PropertyType`, and `Value` parameters of the `New-ItemProperty` cmdlet by implementing the `Microsoft.PowerShell.Commands.RegistryProvider.NewProperty*` method.

`Remove-Item` You can define how your provider will use the values passed to the `Path` and `Recurse` parameters of the `Remove-Item` cmdlet by implementing the `System.Management.Automation.Provider.ContainerCmdletProvider.RemoveItem*` method.

`Remove-ItemProperty` You can define how your provider will use the values passed to the `Path` and `Name` parameters of the `Remove-ItemProperty` cmdlet by implementing the [System.Management.Automation.Provider.IDynamicPropertyCmdletProvider.RemoveProperty*](#) method.

`Rename-Item` cmdlet You can define how your provider will use the values passed to the `Path` and `NewName` parameters of the `Rename-Item` cmdlet by implementing the [System.Management.Automation.Provider.ContainerCmdletProvider.RenameItem*](#) method.

`Rename-ItemProperty` You can define how your provider will use the values passed to the `Path`, `NewName`, and `Name` parameters of the `Rename-ItemProperty` cmdlet by implementing the [System.Management.Automation.Provider.IDynamicPropertyCmdletProvider.RenameProperty*](#) method.

`Set-Content` cmdlet You can define how your provider will use the values passed to the `Path` parameter of the `Set-Content` cmdlet by implementing the [System.Management.Automation.Provider.IContentCmdletProvider.GetContentWriter*](#) method.

`Set-Item` cmdlet You can define how your provider will use the values passed to the `Path` and `Value` parameters of the `Set-Item` cmdlet by implementing the [System.Management.Automation.Provider.ItemCmdletProvider.SetItem*](#) method.

`Set-ItemProperty` cmdlet You can define how your provider will use the values passed to the `Path` and `Value` parameters of the `Set-Item` cmdlet by implementing the [System.Management.Automation.Provider.IPropertyCmdletProvider SetProperty*](#) method.

`Test-Path` cmdlet You can define how your provider will use the values passed to the `Path` parameter of the `Test-Path` cmdlet by implementing the [System.Management.Automation.Provider.ItemCmdletProvider.InvokeDefaultAction*](#) method.

In addition, you cannot specify the characteristics of these parameters, such as whether they are optional or required, nor can you give these parameters an alias or specify any of the validation attributes. In contrast, you can specify parameter characteristics in stand-alone cmdlets by using attributes such as the `Parameters` attribute.

Provider Cmdlet Dynamic Parameters

Dynamic parameters for cmdlet providers are similar to dynamic providers for stand-alone cmdlets. In both cases, the parameters are added to the cmdlet when the user specifies a certain value for one of the default parameters, such as the `path` parameter. However, not all of the static parameters can be used to trigger the addition of dynamic parameters. For more information about dynamic parameters, see [Provider Cmdlet Dynamic Parameters](#).

See Also

[Provider Cmdlet Dynamic Parameters](#)

[Writing a Windows PowerShell Provider](#)

Provider cmdlet dynamic parameters

Article • 01/31/2024

Providers can define dynamic parameters that are added to a provider cmdlet when the user specifies a certain value for one of the static parameters of the cmdlet. For example, a provider can add different dynamic parameters based on what path the user specifies when they call the `Get-Item` or `Set-Item` provider cmdlets.

Dynamic Parameter Methods

Dynamic parameters are defined by implementing one of the dynamic parameter methods, such as the

`System.Management.Automation.Provider.ItemCmdletProvider.GetItemDynamicParameters*` and

`System.Management.Automation.Provider.SetItemDynamicParameters.SetItemDynamicParameters*` methods. These methods return an object that has public properties that are decorated with attributes similar to those of stand-alone cmdlets. Here is an example of an implementation of the

`System.Management.Automation.Provider.ItemCmdletProvider.GetItemDynamicParameters*` method taken from the Certificate provider:

C#

```
protected override object GetItemDynamicParameters(string path)
{
    return new CertificateProviderDynamicParameters();
}
```

Unlike the static parameters of provider cmdlets, you can specify the characteristics of these parameters in the same way that parameters are defined in stand-alone cmdlets. Here is an example of a dynamic parameter class taken from the Certificate provider:

C#

```
internal sealed class CertificateProviderDynamicParameters
{
    /// <summary>
    /// Dynamic parameter the controls whether we only return
    /// code signing certs.
    /// </summary>
    [Parameter()]
    public SwitchParameter CodeSigningCert
    {
```

```

    get
    {
        {
            return codeSigningCert;
        }
    }

    set
    {
        {
            codeSigningCert = value;
        }
    }
}

private SwitchParameter codeSigningCert = new SwitchParameter();
}

```

Dynamic Parameters

Here is a list of the static parameters that can be used to add dynamic parameters.

- `Clear-Content` cmdlet - You can define dynamic parameters that are triggered by the `Path` parameter of the Clear-Clear cmdlet by implementing the `System.Management.Automation.Provider.IContentCmdletProvider.ClearContentDynamicParameters*` method.
- `Clear-Item` cmdlet - You can define dynamic parameters that are triggered by the `Path` parameter of the `Clear-Item` cmdlet by implementing the `System.Management.Automation.Provider.ItemCmdletProvider.ClearItemDynamicParameters*` method.
- `Clear-ItemProperty` cmdlet - You can define dynamic parameters that are triggered by the `Path` parameter of the `Clear-ItemProperty` cmdlet by implementing the `System.Management.Automation.Provider.IPropertyCmdletProvider.ClearPropertyDynamicParameters*` method.
- `Copy-Item` cmdlet - You can define dynamic parameters that are triggered by the `Path`, `Destination`, and `Recurse` parameters of the `Copy-Item` cmdlet by implementing the `System.Management.Automation.Provider.ContainerCmdletProvider.CopyItemDynamicParameters*` method.

- `Get-ChildItem` cmdlet - You can define dynamic parameters that are triggered by the `Path` and `Recurse` parameters of the `Get-ChildItem` cmdlet by implementing the `System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItemsDynamicParameters*` and `System.Management.Automation.Provider.ContainerCmdletProvider.GetChildNamesDynamicParameters*` methods.
- `Get-Content` cmdlet - You can define dynamic parameters that are triggered by the `Path` parameter of the `Get-Content` cmdlet by implementing the `System.Management.Automation.Provider.IContentCmdletProvider.GetContentReaderDynamicParameters*` method.
- `Get-Item` cmdlet - You can define dynamic parameters that are triggered by the `Path` parameter of the `Get-Item` cmdlet by implementing the `System.Management.Automation.Provider.ItemCmdletProvider.GetItemDynamicParameters*` method.
- `Get-ItemProperty` cmdlet - You can define dynamic parameters that are triggered by the `Path` and `Name` parameters of the `Get-ItemProperty` cmdlet by implementing the `System.Management.Automation.Provider.IPropertyCmdletProvider.GetPropertyDynamicParameters*` method.
- `Invoke-Item` cmdlet - You can define dynamic parameters that are triggered by the `Path` parameter of the `Invoke-Item` cmdlet by implementing the `System.Management.Automation.Provider.ItemCmdletProvider.InvokeDefaultActionDynamicParameters*` method.
- `Move-Item` cmdlet - You can define dynamic parameters that are triggered by the `Path` and `Destination` parameters of the `Move-Item` cmdlet by implementing the `System.Management.Automation.Provider.NavigationCmdletProvider.MoveItemDynamicParameters*` method.
- `New-Item` cmdlet - You can define dynamic parameters that are triggered by the `Path`, `ItemType`, and `Value` parameters of the `New-Item` cmdlet by implementing the `System.Management.Automation.Provider.ContainerCmdletProvider.NewItemDynamicParameters*` method.
- `New-ItemProperty` cmdlet - You can define dynamic parameters that are triggered by the `Path`, `Name`, `.PropertyType`, and `Value` parameters of the `New-ItemProperty`

cmdlet by implementing the `System.Management.Automation.Provider.IDynamicPropertyCmdletProvider.NewPropertyDynamicParameters*` method.

- `New-PSDrive` cmdlet - You can define dynamic parameters that are triggered by the `System.Management.Automation.PSDriveInfo` object returned by the `New-PSDrive` cmdlet by implementing the `System.Management.Automation.Provider.DriveCmdletProvider.NewDriveDynamicParameters*` method.
- `Remove-Item` cmdlet - You can define dynamic parameters that are triggered by the `Path` and `Recurse` parameters of the `Remove-Item` cmdlet by implementing the `System.Management.Automation.Provider.ContainerCmdletProvider.RemoveItemDynamicParameters*` method.
- `Remove-ItemProperty` cmdlet - You can define dynamic parameters that are triggered by the `Path` and `Name` parameters of the `Remove-ItemProperty` cmdlet by implementing the `System.Management.Automation.Provider.IDynamicPropertyCmdletProvider.RemovePropertyDynamicParameters*` method.
- `Rename-Item` cmdlet - You can define dynamic parameters that are triggered by the `Path` and `NewName` parameters of the `Rename-Item` cmdlet by implementing the `System.Management.Automation.Provider.ContainerCmdletProvider.RenameItemDynamicParameters*` method.
- `Rename-ItemProperty` - You can define dynamic parameters that are triggered by the `Path`, `Name`, and `NewName` parameters of the `Rename-ItemProperty` cmdlet by implementing the `System.Management.Automation.Provider.IDynamicPropertyCmdletProvider.RenamePropertyDynamicParameters*` method.
- `Set-Content` cmdlet - You can define dynamic parameters that are triggered by the `Path` parameter of the `Set-Content` cmdlet by implementing the `System.Management.Automation.Provider.IContentCmdletProvider.GetContentWriterDynamicParameters*` method.
- `Set-Item` cmdlet - You can define dynamic parameters that are triggered by the `Path` and `Value` parameters of the `Set-Item` cmdlet by implementing the `System.Management.Automation.Provider.ItemCmdletProvider.SetItemDynamicParameters*` method.

- `Set-ItemProperty` cmdlet - You can define dynamic parameters that are triggered by the `Path` and `Value` parameters of the `Set-Item` cmdlet by implementing the `System.Management.Automation.Provider.IPropertyCmdletProvider SetPropertyDynamicParameters*` method.
- `Test-Path` cmdlet - You can define dynamic parameters that are triggered by the `Path` parameter of the `Test-Path` cmdlet by implementing the `System.Management.Automation.Provider.ItemCmdletProvider.InvokeDefaultActionDynamicParameters*` method.

See Also

[Writing a Windows PowerShell Provider](#)

Writing an item provider

Article • 03/24/2025

This topic describes how to implement the methods of a Windows PowerShell provider that access and manipulate items in the data store. To be able to access items, a provider must derive from the [System.Management.Automation.Provider.ItemCmdletProvider](#) class.

The provider in the examples in this topic uses an Access database as its data store. There are several helper methods and classes that are used to interact with the database. For the complete sample that includes the helper methods, see [AccessDBProviderSample03](#)

For more information about Windows PowerShell providers, see [Windows PowerShell Provider Overview](#).

Implementing item methods

The [System.Management.Automation.Provider.ItemCmdletProvider](#) class exposes several methods that can be used to access and manipulate the items in a data store. For a complete list of these methods, see [ItemCmdletProvider Methods](#). In this example, we will implement four of these methods.

[System.Management.Automation.Provider.ItemCmdletProvider.GetItem*](#) gets an item at a specified path.

[System.Management.Automation.Provider.ItemCmdletProvider.SetItem*](#) sets the value of the specified item.

[System.Management.Automation.Provider.ItemCmdletProvider.ItemExists*](#) checks whether an item exists at the specified path.

[System.Management.Automation.Provider.ItemCmdletProvider.IsValidPath*](#) checks a path to see if it maps to a location in the data store.

ⓘ Note

This topic builds on the information in [Windows PowerShell Provider QuickStart](#).

This topic does not cover the basics of how to set up a provider project, or how to implement the methods inherited from the

[System.Management.Automation.Provider.DriveCmdletProvider](#) class that create and remove drives.

Declaring the provider class

Declare the provider to derive from the [System.Management.Automation.Provider.ItemCmdletProvider](#) class, and decorate it with the [System.Management.Automation.Provider.CmdletProviderAttribute](#).

C#

```
[CmdletProvider("AccessDB", ProviderCapabilities.None)]  
  
public class AccessDBProvider : ItemCmdletProvider  
{  
  
}
```

Implementing GetItem

The [System.Management.Automation.Provider.ItemCmdletProvider.GetItem*](#) is called by the PowerShell engine when a user calls the [Microsoft.PowerShell.Commands.GetItemCommand](#) cmdlet on your provider. The method returns the item at the specified path. In the Access database example, the method checks whether the item is the drive itself, a table in the database, or a row in the database. The method sends the item to the PowerShell engine by calling the [System.Management.Automation.Provider.CmdletProvider.WriteItemObject*](#) method.

C#

```
protected override void GetItem(string path)  
{  
    // check if the path represented is a drive  
    if (PathIsDrive(path))  
    {  
        WriteItemObject(this.PSDriveInfo, path, true);  
        return;  
    } // if (PathIsDrive...  
  
    // Get table name and row information from the path and do  
    // necessary actions  
    string tableName;  
    int rowCount;  
  
    PathType type = GetNamesFromPath(path, out tableName, out  
    rowCount);  
  
    if (type == PathType.Table)  
    {  
        DatabaseTableInfo table = GetTable(tableName);  
    }  
}
```

```

        WriteItemObject(table, path, true);
    }
    else if (type == PathType.Row)
    {
        DatabaseRowInfo row = GetRow(tableName, lineNumber);
        WriteItemObject(row, path, false);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }

}

```

Implementing SetItem

The [System.Management.Automation.Provider.ItemCmdletProvider.SetItem*](#) method is called by the PowerShell engine calls when a user calls the [Microsoft.PowerShell.Commands.SetItemCommand](#) cmdlet. It sets the value of the item at the specified path.

In the Access database example, it makes sense to set the value of an item only if that item is a row, so the method throws [NotSupportedException](#) when the item is not a row.

C#

```

protected override void SetItem(string path, object values)
{
    // Get type, table name and row number from the path specified
    string tableName;
    int lineNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type != PathType.Row)
    {
        WriteError(new ErrorRecord(new NotSupportedException(
            "SetNotSupported"), "", 
        ErrorCategory.InvalidOperation, path));

        return;
    }

    // Get in-memory representation of table
    OdbcDataAdapter da = GetAdapterForTable(tableName);

    if (da == null)
    {
        return;
    }
}

```

```

        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        if (rowNumber >= table.Rows.Count)
        {
            // The specified row number has to be available. If not
            // NewItem has to be used to add a new row
            throw new ArgumentException("Row specified is not
available");
        } // if (rowNum...

        string[] colValues = (values as string).Split(',');
        // set the specified row
        DataRow row = table.Rows[rowNumber];

        for (int i = 0; i < colValues.Length; i++)
        {
            row[i] = colValues[i];
        }

        // Update the table
        if (ShouldProcess(path, "SetItem"))
        {
            da.Update(ds, tableName);
        }
    }
}

```

Implementing ItemExists

The [System.Management.Automation.Provider.ItemCmdletProvider.ItemExists*](#) method is called by the PowerShell engine when a user calls the [Microsoft.PowerShell.Commands.TestPathCommand](#) cmdlet. The method determines whether there is an item at the specified path. If the item does exist, the method passes it back to the PowerShell engine by calling [System.Management.Automation.Provider.CmdletProvider.WriteItemObject*](#).

C#

```

protected override bool ItemExists(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        return true;
    }

    // Obtain type, table name and row number from path
}

```

```

        string tableName;
        int rowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        DatabaseTableInfo table = GetTable(tableName);

        if (type == PathType.Table)
        {
            // if specified path represents a table then
DatabaseTableInfo
            // object for the same should exist
            if (table != null)
            {
                return true;
            }
        }
        else if (type == PathType.Row)
        {
            // if specified path represents a row then DatabaseTableInfo
should
            // exist for the table and then specified row number must be
within
            // the maximum row count in the table
            if (table != null && rowNumber < table.RowCount)
            {
                return true;
            }
        }

        return false;

    }

```

Implementing IsValidPath

The [System.Management.Automation.Provider.ItemCmdletProvider.IsValidPath*](#) method checks whether the specified path is syntactically valid for the current provider. It does not check whether an item exists at the path.

C#

```

protected override bool IsValidPath(string path)
{
    bool result = true;

    // check if the path is null or empty
    if (String.IsNullOrEmpty(path))
    {
        result = false;
    }
}
```

```
}

// convert all separators in the path to a uniform one
path = NormalizePath(path);

// split the path into individual chunks
string[] pathChunks = path.Split(pathSeparator.ToCharArray());

foreach (string pathChunk in pathChunks)
{
    if (pathChunk.Length == 0)
    {
        result = false;
    }
}
return result;
}
```

Next steps

A typical real-world provider is capable of supporting items that contain other items, and of moving items from one path to another within the drive. For an example of a provider that supports containers, see [Writing a container provider](#). For an example of a provider that supports moving items, see [Writing a navigation provider](#).

See Also

[Writing a container provider](#)

[Writing a navigation provider](#)

[Windows PowerShell Provider Overview](#)

Writing a container provider

Article • 03/24/2025

This topic describes how to implement the methods of a Windows PowerShell provider that support items that contain other items, such as folders in the FileSystem provider.

To be able to support containers, a provider must derive from the

[System.Management.Automation.Provider.ContainerCmdletProvider](#) class.

The provider in the examples in this topic uses an Access database as its data store.

There are several helper methods and classes that are used to interact with the

database. For the complete sample that includes the helper methods, see

[AccessDBProviderSample04](#).

For more information about Windows PowerShell providers, see [Windows PowerShell Provider Overview](#).

Implementing container methods

The [System.Management.Automation.Provider.ContainerCmdletProvider](#) class implements methods that support containers, and create, copy, and remove items. For a complete list of these methods, see [System.Management.Automation.Provider.ContainerCmdletProvider](#).

ⓘ Note

This topic builds on the information in [Windows PowerShell Provider QuickStart](#).

This topic does not cover the basics of how to set up a provider project, or how to implement the methods inherited from the

[System.Management.Automation.Provider.DriveCmdletProvider](#) class that create and remove drives. This topic also does not cover how to implement methods exposed by the [System.Management.Automation.Provider.ItemCmdletProvider](#) class. For an example that shows how to implement item cmdlets, see [Writing an item provider](#).

Declaring the provider class

Declare the provider to derive from the

[System.Management.Automation.Provider.ContainerCmdletProvider](#) class, and decorate it with the [System.Management.Automation.Provider.CmdletProviderAttribute](#).

C#

```
[CmdletProvider("AccessDB", ProviderCapabilities.None)]
public class AccessDBProvider : ContainerCmdletProvider
{
}
```

Implementing GetChildItems

The PowerShell engine calls the

[System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems*](#) method when a user calls the [Microsoft.PowerShell.Commands.GetChildItemCommand](#) cmdlet. This method gets the items that are the children of the item at the specified path.

In the Access database example, the behavior of the

[System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems*](#) method depends on the type of the specified item. If the item is the drive, then the children are tables, and the method returns the set of tables from the database. If the specified item is a table, then the children are the rows of that table. If the item is a row, then it has no children, and the method returns that row only. All child items are sent back to the PowerShell engine by the

[System.Management.Automation.Provider.CmdletProvider.WriteItemObject*](#) method.

C#

```
protected override void GetChildItems(string path, bool recurse)
{
    // If path represented is a drive then the children in the path
    // are
    // tables. Hence all tables in the drive represented will have to
    // be
    // returned
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table, path, true);

            // if the specified item exists and recurse has been set
            // then
            // all child items within it have to be obtained as well
            if (ItemExists(path) && recurse)
            {
                GetChildItems(path + pathSeparator + table.Name,
                recurse);
            }
        }
    }
}
```

```

        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get the table name, row number and type of path from the
        // path specified
        string tableName;
        int rowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Table)
        {
            // Obtain all the rows within the table
            foreach (DatabaseRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row, path + pathSeparator +
row.RowNumber,
                                false);
            } // foreach (DatabaseRowInfo...
        }
        else if (type == PathType.Row)
        {
            // In this case the user has directly specified a row,
hence
            // just give that particular row
            DatabaseRowInfo row = GetRow(tableName, rowNumber);
            WriteItemObject(row, path + pathSeparator +
row.RowNumber,
                                false);
        }
        else
        {
            // In this case, the path specified is not valid
            ThrowTerminatingInvalidOperationException(path);
        }
    } // else
}

```

Implementing GetChildNames

The

[System.Management.Automation.Provider.ContainerCmdletProvider.GetChildNames*](#)
method is similar to the

[System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems*](#)
method, except that it returns only the name property of the items, and not the items
themselves.

```

protected override void GetChildNames(string path,
                                     ReturnContainers returnContainers)
{
    // If the path represented is a drive, then the child items are
    // tables. get the names of all the tables in the drive.
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table.Name, path, true);
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get type, table name and row number from path specified
        string tableName;
        int rowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Table)
        {
            // Get all the rows in the table and then write out the
            // row numbers.
            foreach (DatabaseRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row.RowNumber, path, false);
            } // foreach (DatabaseRowInfo...
        }
        else if (type == PathType.Row)
        {
            // In this case the user has directly specified a row,
            hence
            // just give that particular row
            DatabaseRowInfo row = GetRow(tableName, rowNumber);

            WriteItemObject(row.RowNumber, path, false);
        }
        else
        {
            ThrowTerminatingInvalidOperationException(path);
        }
    } // else
}

```

Implementing NewItem

The [System.Management.Automation.Provider.ContainerCmdletProvider.NewItem](#)* method creates a new item of the specified type at the specified path. The PowerShell

engine calls this method when a user calls the [Microsoft.PowerShell.Commands.NewItemCommand](#) cmdlet.

In this example, the method implements logic to determine that the path and type match. That is, only a table can be created directly under the drive (the database), and only a row can be created under a table. If the specified path and item type don't match in this way, the method throws an exception.

C#

```
protected override void NewItem(string path, string type,
                               object newItemValue)
{
    string tableName;
    int rowNumber;

    PathType pt = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (pt == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }

    // Check if type is either "table" or "row", if not throw an
    // exception
    if (!String.Equals(type, "table",
 StringComparison.OrdinalIgnoreCase)
        && !String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
    {
        WriteError(new ErrorRecord
            (new ArgumentException("Type must be either
a table or row"),
             "CannotCreateSpecifiedObject",
             ErrorCategory.InvalidArgument,
             path
            )
        );
    }

    throw new ArgumentException("This provider can only create
items of type \"table\" or \"row\"");
}

// Path type is the type of path of the container. So if a drive
// is specified, then a table can be created under it and if a
table
// is specified, then a row can be created under it. For the sake
of
// completeness, if a row is specified, then if the row specified
by
// the path does not exist, a new row is created. However, the
```

```

row
    // number may not match as the row numbers only get incremented
based
    // on the number of rows

    if (PathIsDrive(path))
    {
        if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
        {
            // Execute command using ODBC connection to create a
table
            try
            {
                // create the table using an sql statement
                string newTableName = newItemValue.ToString();

                if (!TableNameIsValid(newTableName))
                {
                    return;
                }
                string sql = "create table " + newTableName
                            + " (ID INT)";

                // Create the table using the Odbc connection from
the
                // drive.
                AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

                if (di == null)
                {
                    return;
                }
                OdbcConnection connection = di.Connection;

                if (ShouldProcess(newTableName, "create"))
                {
                    OdbcCommand cmd = new OdbcCommand(sql,
connection);
                    cmd.ExecuteScalar();
                }
            }
            catch (Exception ex)
            {
                WriteError(new ErrorRecord(ex,
"CannotCreateSpecifiedTable",
ErrorCategory.InvalidOperation, path)
);
            }
        } // if (String...
        else if (String.Equals(type, "row",
StringComparison.OrdinalIgnoreCase))
        {
            throw new

```

```

                    ArgumentException("A row cannot be created under a
database, specify a path that represents a Table");
                }
            } // if (PathIsDrive...
        else
        {
            if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
            {
                if (rowNumber < 0)
                {
                    throw new
                        ArgumentException("A table cannot be created
within another table, specify a path that represents a database");
                }
                else
                {
                    throw new
                        ArgumentException("A table cannot be created
inside a row, specify a path that represents a database");
                }
            } //if (String.Equals....
            // if path specified is a row, create a new row
            else if (String.Equals(type, "row",
StringComparison.OrdinalIgnoreCase))
            {
                // The user is required to specify the values to be
inserted
                // into the table in a single string separated by commas
                string value = newItemValue as string;

                if (String.IsNullOrEmpty(value))
                {
                    throw new
                        ArgumentException("Value argument must have comma
separated values of each column in a row");
                }
                string[] rowValues = value.Split(',');
            }

            OdbcDataAdapter da = GetAdapterForTable(tableName);

            if (da == null)
            {
                return;
            }

            DataSet ds = GetDataSetForTable(da, tableName);
            DataTable table = GetDataTable(ds, tableName);

            if (rowValues.Length != table.Columns.Count)
            {
                string message =
                    String.Format(CultureInfo.CurrentCulture,
                    "The table has {0} columns and
the value specified must have so many comma separated values",

```

```

                table.Columns.Count);

            throw new ArgumentException(message);
        }

        if (!Force && (rowNumber >=0 && rowNumber <
table.Rows.Count))
        {
            string message =
String.Format(CultureInfo.CurrentCulture,
                                         "The row {0} already
exists. To create a new row specify row number as {1}, or specify path to a
table, or use the -Force parameter",
                                         rowNumber,
table.Rows.Count);

            throw new ArgumentException(message);
        }

        if (rowNumber > table.Rows.Count)
        {
            string message =
String.Format(CultureInfo.CurrentCulture,
                                         "To create a new row specify row
number as {0}, or specify path to a table",
                                         table.Rows.Count);

            throw new ArgumentException(message);
        }

        // Create a new row and update the row with the input
        // provided by the user
        DataRow row = table.NewRow();
        for (int i = 0; i < rowValues.Length; i++)
        {
            row[i] = rowValues[i];
        }
        table.Rows.Add(row);

        if (ShouldProcess(tableName, "update rows"))
        {
            // Update the table from memory back to the data
source
            da.Update(ds, tableName);
        }

    }// else if (String...
}// else ...

}

```

Implementing CopyItem

The [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItem](#) copies the specified item to the specified path. The PowerShell engine calls this method when a user calls the [Microsoft.PowerShell.Commands.CopyItemCommand](#) cmdlet. This method can also be recursive, copying all of the items children in addition to the item itself.

Similarly to the

[System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*](#) method, this method performs logic to make sure that the specified item is of the correct type for the path to which it is being copied. For example, if the destination path is a table, the item to be copied must be a row.

C#

```
protected override void CopyItem(string path, string copyPath, bool recurse)
{
    string tableName, copyTableName;
    int rowCount, copyRowCount;

    PathType type = GetNamesFromPath(path, out tableName, out
rowCount);
    PathType copyType = GetNamesFromPath(copyPath, out copyTableName,
out copyRowCount);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidOperationException(path);
    }

    if (copyType == PathType.Invalid)
    {
        ThrowTerminatingInvalidOperationException(copyPath);
    }

    // Get the table and the table to copy to
    OdbcDataAdapter da = GetAdapterForTable(tableName);
    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    OdbcDataAdapter cda = GetAdapterForTable(copyTableName);
    if (cda == null)
    {
        return;
    }

    DataSet cds = GetDataSetForTable(cda, copyTableName);
```

```

DataTable copyTable = GetDataTable(cds, copyTableName);

// if source represents a table
if (type == PathType.Table)
{
    // if copyPath does not represent a table
    if (copyType != PathType.Table)
    {
        ArgumentException e = new ArgumentException("Table can
only be copied on to another table location");

        WriteError(new ErrorRecord(e, "PathNotValid",
            ErrorCategory.InvalidArgument, copyPath));

        throw e;
    }

    // if table already exists then Force parameter should be set
    // to force a copy
    if (!Force && GetTable(copyTableName) != null)
    {
        throw new ArgumentException("Specified path already
exists");
    }

    for (int i = 0; i < table.Rows.Count; i++)
    {
        DataRow row = table.Rows[i];
        DataRow copyRow = copyTable.NewRow();

        copyRow.ItemArray = row.ItemArray;
        copyTable.Rows.Add(copyRow);
    }
} // if (type == ...
// if source represents a row
else
{
    if (copyType == PathType.Row)
    {
        if (!Force && (copyRowNumber < copyTable.Rows.Count))
        {
            throw new ArgumentException("Specified path already
exists.");
        }

        DataRow row = table.Rows[rowNumber];
        DataRow copyRow = null;

        if (copyRowNumber < copyTable.Rows.Count)
        {
            // copy to an existing row
            copyRow = copyTable.Rows[copyRowNumber];
            copyRow.ItemArray = row.ItemArray;
            copyRow[0] = GetNextID(copyTable);
        }
    }
}

```

```

        else if (copyRowNumber == copyTable.Rows.Count)
    {
        // copy to the next row in the table that will
        // be created
        copyRow = copyTable.NewRow();
        copyRow.ItemArray = row.ItemArray;
        copyRow[0] = GetNextID(copyTable);
        copyTable.Rows.Add(copyRow);
    }
    else
    {
        // attempting to copy to a nonexistent row or a row
        // that cannot be created now - throw an exception
        string message =
String.Format(CultureInfo.CurrentCulture,
                           "The item cannot be specified
to the copied row. Specify row number as {0}, or specify a path to the
table.",

                           table.Rows.Count);

        throw new ArgumentException(message);
    }
}
else
{
    // destination path specified represents a table,
    // create a new row and copy the item
    DataRow copyRow = copyTable.NewRow();
    copyRow.ItemArray = table.Rows[rowNumber].ItemArray;
    copyRow[0] = GetNextID(copyTable);
    copyTable.Rows.Add(copyRow);
}
}

if (ShouldProcess(copyTableName, "CopyItems"))
{
    cda.Update(cds, copyTableName);
}

} //CopyItem

```

Implementing RemoveItem

The [System.Management.Automation.Provider.ContainerCmdletProvider.RemoveItem*](#) method removes the item at the specified path. The PowerShell engine calls this method when a user calls the [Microsoft.PowerShell.Commands.RemoveItemCommand](#) cmdlet.

C#

```

protected override void RemoveItem(string path, bool recurse)
{

```

```

    string tableName;
    int rowNumber = 0;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        // if recurse flag has been specified, delete all the rows as
well
        if (recurse)
        {
            OdbcDataAdapter da = GetAdapterForTable(tableName);
            if (da == null)
            {
                return;
            }

            DataSet ds = GetDataSetForTable(da, tableName);
            DataTable table = GetDataTable(ds, tableName);

            for (int i = 0; i < table.Rows.Count; i++)
            {
                table.Rows[i].Delete();
            }

            if (ShouldProcess(path, "RemoveItem"))
            {
                da.Update(ds, tableName);
                RemoveTable(tableName);
            }
        }//if (recurse...
        else
        {
            // Remove the table
            if (ShouldProcess(path, "RemoveItem"))
            {
                RemoveTable(tableName);
            }
        }
    }
    else if (type == PathType.Row)
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);
        if (da == null)
        {
            return;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        table.Rows[rowNumber].Delete();

        if (ShouldProcess(path, "RemoveItem"))

```

```
        {
            da.Update(ds, tableName);
        }
    }
else
{
    ThrowTerminatingInvalidOperationException(path);
}

}
```

Next steps

A typical real-world provider is capable of moving items from one path to another within the drive. For an example of a provider that supports moving items, see [Writing a navigation provider](#).

See Also

[Writing a navigation provider](#)

[Windows PowerShell Provider Overview](#)

Writing a navigation provider

Article • 03/24/2025

This topic describes how to implement the methods of a Windows PowerShell provider that support nested containers (multi-level data stores), moving items, and relative paths. A navigation provider must derive from the [System.Management.Automation.Provider.NavigationCmdletProvider](#) class.

The provider in the examples in this topic uses an Access database as its data store. There are several helper methods and classes that are used to interact with the database. For the complete sample that includes the helper methods, see [AccessDBProviderSample05](#).

For more information about Windows PowerShell providers, see [Windows PowerShell Provider Overview](#).

Implementing navigation methods

The [System.Management.Automation.Provider.NavigationCmdletProvider](#) class implements methods that support nested containers, relative paths, and moving items. For a complete list of these methods, see [NavigationCmdletProvider Methods](#).

Note

This topic builds on the information in [Windows PowerShell Provider QuickStart](#). This topic does not cover the basics of how to set up a provider project, or how to implement the methods inherited from the [System.Management.Automation.Provider.DriveCmdletProvider](#) class that create and remove drives. This topic also does not cover how to implement methods exposed by the [System.Management.Automation.Provider.ItemCmdletProvider](#) or [System.Management.Automation.Provider.ContainerCmdletProvider](#) classes. For an example that shows how to implement item cmdlets, see [Writing an item provider](#). For an example that shows how to implement container cmdlets, see [Writing a container provider](#).

Declaring the provider class

Declare the provider to derive from the [System.Management.Automation.Provider.NavigationCmdletProvider](#) class, and

decorate it with the [System.Management.Automation.Provider.CmdletProviderAttribute](#).

```
[CmdletProvider("AccessDB", ProviderCapabilities.None)]
public class AccessDBProvider : NavigationCmdletProvider
{
}
```

Implementing IsItemContainer

The [System.Management.Automation.Provider.NavigationCmdletProvider.IsItemContainer*](#) method checks whether the item at the specified path is a container.

C#

```
protected override bool IsItemContainer(string path)
{
    if (PathIsDrive(path))
    {
        return true;
    }

    string[] pathChunks = ChunkPath(path);
    string tableName;
    int rowCount;

    PathType type = GetNamesFromPath(path, out tableName, out
rowCount);

    if (type == PathType.Table)
    {
        foreach (DatabaseTableInfo ti in GetTables())
        {
            if (string.Equals(ti.Name, tableName,
StringComparison.OrdinalIgnoreCase))
            {
                return true;
            }
        } // foreach (DatabaseTableInfo...
    } // if (pathChunks...

    return false;
}
```

Implementing GetChildName

The

[System.Management.Automation.Provider.NavigationCmdletProvider.GetChildNames*](#) method gets the name property of the child item at the specified path. If the item at the specified path is not a child of a container, then this method should return the path.

C#

```
protected override string GetChildName(string path)
{
    if (PathIsDrive(path))
    {
        return path;
    }

    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        return tableName;
    }
    else if (type == PathType.Row)
    {
        return rowNumber.ToString(CultureInfo.CurrentCulture);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }

    return null;
}
```

Implementing GetParentPath

The

[System.Management.Automation.Provider.NavigationCmdletProvider.GetParentPath*](#) method gets the path of the parent of the item at the specified path. If the item at the specified path is the root of the data store (so it has no parent), then this method should return the root path.

C#

```
protected override string GetParentPath(string path, string root)
{
    // If root is specified then the path has to contain
```

```

// the root. If not nothing should be returned
if (!String.IsNullOrEmpty(root))
{
    if (!path.Contains(root))
    {
        return null;
    }
}

return path.Substring(0, path.LastIndexOf(pathSeparator,
 StringComparison.OrdinalIgnoreCase));
}

```

Implementing MakePath

The [System.Management.Automation.Provider.NavigationCmdletProvider.MakePath*](#) method joins a specified parent path and a specified child path to create a provider-internal path (for information about path types that providers can support, see [Windows PowerShell Provider Overview](#)). The PowerShell engine calls this method when a user calls the [Microsoft.PowerShell.Commands.JoinPathCommand](#) cmdlet.

C#

```

protected override string MakePath(string parent, string child)
{
    string result;

    string normalParent = NormalizePath(parent);
    normalParent = RemoveDriveFromPath(normalParent);
    string normalChild = NormalizePath(child);
    normalChild = RemoveDriveFromPath(normalChild);

    if (String.IsNullOrEmpty(normalParent) &&
String.IsNullOrEmpty(normalChild))
    {
        result = String.Empty;
    }
    else if (String.IsNullOrEmpty(normalParent) &&
!String.IsNullOrEmpty(normalChild))
    {
        result = normalChild;
    }
    else if (!String.IsNullOrEmpty(normalParent) &&
String.IsNullOrEmpty(normalChild))
    {
        if (normalParent.EndsWith(pathSeparator,
StringComparison.OrdinalIgnoreCase))
        {
            result = normalParent;
        }
        else
    }
}

```

```

        {
            result = normalParent + pathSeparator;
        }
    } // else if (!String...
else
{
    if (!normalParent.Equals(String.Empty) &&
        !normalParent.EndsWith(pathSeparator,
StringComparison.OrdinalIgnoreCase))
    {
        result = normalParent + pathSeparator;
    }
    else
    {
        result = normalParent;
    }

    if (normalChild.StartsWith(pathSeparator,
StringComparison.OrdinalIgnoreCase))
    {
        result += normalChild.Substring(1);
    }
    else
    {
        result += normalChild;
    }
} // else

return result;
}

```

Implementing NormalizeRelativePath

The

[System.Management.Automation.Provider.NavigationCmdletProvider.NormalizeRelativePath](#)* method takes `path` and `basepath` parameters, and returns a normalized path that is equivalent to the `path` parameter and relative to the `basepath` parameter.

C#

```

protected override string NormalizeRelativePath(string path,
                                                string basepath)
{
    // Normalize the paths first
    string normalPath = NormalizePath(path);
    normalPath = RemoveDriveFromPath(normalPath);
    string normalBasePath = NormalizePath(basepath);
    normalBasePath = RemoveDriveFromPath(normalBasePath);

    if (String.IsNullOrEmpty(normalBasePath))
    {

```

```

        return normalPath;
    }
    else
    {
        if (!normalPath.Contains(normalBasePath))
        {
            return null;
        }

        return normalPath.Substring(normalBasePath.Length +
pathSeparator.Length);
    }
}

```

Implementing MoveItem

The [System.Management.Automation.Provider.NavigationCmdletProvider.MoveItem*](#) method moves an item from the specified path to the specified destination path. The PowerShell engine calls this method when a user calls the [Microsoft.PowerShell.Commands.MoveItemCommand](#) cmdlet.

C#

```

protected override void MoveItem(string path, string destination)
{
    // Get type, table name and rowNumber from the path
    string tableName, destTableName;
    int rowNumber, destRowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    PathType destType = GetNamesFromPath(destination, out
destTableName,
                                         out destRowNumber);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidOperationException(path);
    }

    if (destType == PathType.Invalid)
    {
        ThrowTerminatingInvalidOperationException(destination);
    }

    if (type == PathType.Table)
    {
        ArgumentException e = new ArgumentException("Move not
supported for tables");

```

```

        WriteError(new ErrorRecord(e, "MoveNotSupported",
            ErrorCategory.InvalidArgument, path));

    throw e;
}
else
{
    OdbcDataAdapter da = GetAdapterForTable(tableName);
    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    OdbcDataAdapter dda = GetAdapterForTable(destTableName);
    if (dda == null)
    {
        return;
    }

    DataSet dds = GetDataSetForTable(dda, destTableName);
    DataTable destTable = GetDataTable(dds, destTableName);
    DataRow row = table.Rows[rowNumber];

    if (destType == PathType.Table)
    {
        DataRow destRow = destTable.NewRow();

        destRow.ItemArray = row.ItemArray;
    }
    else
    {
        DataRow destRow = destTable.Rows[destRowNumber];

        destRow.ItemArray = row.ItemArray;
    }

    // Update the changes
    if (ShouldProcess(path, "MoveItem"))
    {
        WriteItemObject(row, path, false);
        dda.Update(dds, destTableName);
    }
}
}

```

See Also

[Writing a container provider](#)

Windows PowerShell Provider Overview

Provider Samples

Article • 03/24/2025

This section includes samples of providers that access a Microsoft Access database. These samples include provider classes that derive from all the base provider classes.

In This Section

This section includes the following topics:

[AccessDBProviderSample01 Sample](#) This sample shows how to declare the provider class that derives directly from the [System.Management.Automation.Provider.CmdletProvider](#) class. It is included here only for completeness.

[AccessDBProviderSample02](#) This sample shows how to overwrite the [System.Management.Automation.Provider.DriveCmdletProvider.NewDrive*](#) and [System.Management.Automation.Provider.DriveCmdletProvider.RemoveDrive*](#) methods to support calls to the `New-PSDrive` and `Remove-PSDrive` cmdlets. The provider class in this sample derives from the [System.Management.Automation.Provider.DriveCmdletProvider](#) class.

[AccessDBProviderSample03](#) This sample shows how to overwrite the [System.Management.Automation.Provider.ItemCmdletProvider.GetItem*](#) and [System.Management.Automation.Provider.ItemCmdletProvider.SetItem*](#) methods to support calls to the `Get-Item` and `Set-Item` cmdlets. The provider class in this sample derives from the [System.Management.Automation.Provider.ItemCmdletProvider](#) class.

[AccessDBProviderSample04](#) This sample shows how to overwrite container methods to support calls to the `Copy-Item`, `Get-ChildItem`, `New-Item`, and `Remove-Item` cmdlets. These methods should be implemented when the data store contains items that are containers. A container is a group of child items under a common parent item. The provider class in this sample derives from the [System.Management.Automation.Provider.ContainerCmdletProvider](#) class.

[AccessDBProviderSample05](#) This sample shows how to overwrite container methods to support calls to the `Move-Item` and `Join-Path` cmdlets. These methods should be implemented when the user needs to move items within a container and if the data store contains nested containers. The provider class in this sample derives from the [System.Management.Automation.Provider.NavigationCmdletProvider](#) class.

[AccessDBProviderSample06](#) This sample shows how to overwrite content methods to support calls to the `Clear-Content`, `Get-Content`, and `Set-Content` cmdlets. These methods should be implemented when the user needs to manage the content of the items in the data store. The provider class in this sample derives from the `System.Management.Automation.Provider.NavigationCmdletProvider` class, and it implements the `System.Management.Automation.Provider.IContentCmdletProvider` interface.

See Also

[Writing a Windows PowerShell Provider](#)

AccessDBProviderSample01

Article • 03/24/2025

This sample shows how to declare a provider class that derives directly from the [System.Management.Automation.Provider.CmdletProvider](#) class. It is included here only for completeness.

Demonstrates

Important

Your provider class will most likely derive from one of the following classes and possibly implement other provider interfaces:

- [System.Management.Automation.Provider.ItemCmdletProvider](#) class. See [AccessDBProviderSample03](#).
- [System.Management.Automation.Provider.ContainerCmdletProvider](#) class. See [AccessDBProviderSample04](#).
- [System.Management.Automation.Provider.NavigationCmdletProvider](#) class. See [AccessDBProviderSample05](#).

For more information about choosing which provider class to derive from based on provider features, see [Designing Your Windows PowerShell Provider](#).

This sample demonstrates the following:

- Declaring the `CmdletProvider` attribute.
- Defining a provider class that derives directly from the [System.Management.Automation.Provider.CmdletProvider](#) class.

Example

This sample shows how to define a provider class and how to declare the `CmdletProvider` attribute.

C#

```
using System.Management.Automation;
using System.Management.Automation.Provider;
```

```
using System.ComponentModel;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

    /// <summary>
    /// Simple provider.
    /// </summary>
    [CmdletProvider("AccessDB", ProviderCapabilities.None)]
    public class AccessDBProvider : CmdletProvider
    {

    }

    #endregion AccessDBProvider
}
```

See Also

[System.Management.Automation.Provider.ItemCmdletProvider](#)

[System.Management.Automation.Provider.ContainerCmdletProvider](#)

[System.Management.Automation.Provider.NavigationCmdletProvider](#)

[Designing Your Windows PowerShell Provider](#)

AccessDBProviderSample02

Article • 03/24/2025

This sample shows how to overwrite the `System.Management.Automation.Provider.DriveCmdletProvider.NewDrive*` and `System.Management.Automation.Provider.DriveCmdletProvider.RemoveDrive*` methods to support calls to the `New-PSDrive` and `Remove-PSDrive` cmdlets. The provider class in this sample derives from the `System.Management.Automation.Provider.DriveCmdletProvider` class.

Demonstrates

Important

Your provider class will most likely derive from one of the following classes and possibly implement other provider interfaces:

- [System.Management.Automation.Provider.ItemCmdletProvider](#) class. See [AccessDBProviderSample03](#).
- [System.Management.Automation.Provider.ContainerCmdletProvider](#) class. See [AccessDBProviderSample04](#).
- [System.Management.Automation.Provider.NavigationCmdletProvider](#) class. See [AccessDBProviderSample05](#).

For more information about choosing which provider class to derive from based on provider features, see [Designing Your Windows PowerShell Provider](#).

This sample demonstrates the following:

- Declaring the `CmdletProvider` attribute.
- Defining a provider class that derives from the `System.Management.Automation.Provider.DriveCmdletProvider` class.
- Overwriting the `System.Management.Automation.Provider.DriveCmdletProvider.NewDrive*` method to support creating new drives. (This sample does not show how to add dynamic parameters to the `New-PSDrive` cmdlet.)

- Overwriting the [System.Management.Automation.Provider.DriveCmdletProvider.RemoveDrive*](#) method to support removing existing drives.

Example

This sample shows how to overwrite the [System.Management.Automation.Provider.DriveCmdletProvider.NewDrive*](#) and [System.Management.Automation.Provider.DriveCmdletProvider.RemoveDrive*](#) methods. For this sample provider, when a drive is created its connection information is stored in an `AccessDBPsDriveInfo` object.

C#

```
using System;
using System.IO;
using System.Data;
using System.Data.Odbc;
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.ComponentModel;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

        /// <summary>
        /// A PowerShell Provider which acts upon a access data store.
        /// </summary>
        /// <remarks>
        /// This example only demonstrates the drive overrides
        /// </remarks>
        [CmdletProvider("AccessDB", ProviderCapabilities.None)]
        public class AccessDBProvider : DriveCmdletProvider
    {
        #region Drive Manipulation

            /// <summary>
            /// Create a new drive. Create a connection to the database file and
            /// set
            /// the Connection property in the PSDriveInfo.
            /// </summary>
            /// <param name="drive">
            /// Information describing the drive to add.
            /// </param>
            /// <returns>The added drive.</returns>
            protected override PSDriveInfo NewDrive(PSDriveInfo drive)
            {
                // check if drive object is null
                if (drive == null)
```

```

        {
            WriteError(new ErrorRecord(
                new ArgumentNullException("drive"),
                "NullDrive",
                ErrorCategory.InvalidArgument,
                null)
            );
        }

        return null;
    }

    // check if drive root is not null or empty
    // and if its an existing file
    if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root)
== false))
    {
        WriteError(new ErrorRecord(
            new ArgumentException("drive.Root"),
            "NoRoot",
            ErrorCategory.InvalidArgument,
            drive)
        );
    }

    return null;
}

// create a new drive and create an ODBC connection to the new
drive
AccessDBPSDriveInfo accessDBPSDriveInfo = new
AccessDBPSDriveInfo(drive);

OdbcConnectionStringBuilder builder = new
OdbcConnectionStringBuilder();

builder.Driver = "Microsoft Access Driver (*.mdb)";
builder.Add("DBQ", drive.Root);

OdbcConnection conn = new
OdbcConnection(builder.ConnectionString);
conn.Open();
accessDBPSDriveInfo.Connection = conn;

return accessDBPSDriveInfo;
} // NewDrive

/// <summary>
/// Removes a drive from the provider.
/// </summary>
/// <param name="drive">The drive to remove.</param>
/// <returns>The drive removed.</returns>
protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {

```

```

        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            drive)
        );
    }

    // close ODBC connection to the drive
    AccessDBPSDriveInfo accessDBPSDriveInfo = drive as
AccessDBPSDriveInfo;

    if (accessDBPSDriveInfo == null)
    {
        return null;
    }
    accessDBPSDriveInfo.Connection.Close();

    return accessDBPSDriveInfo;
} // RemoveDrive

#endregion Drive Manipulation

} // AccessDBProvider

#endregion AccessDBProvider

#region AccessDBPSDriveInfo

/// <summary>
/// Any state associated with the drive should be held here.
/// In this case, it's the connection to the database.
/// </summary>
internal class AccessDBPSDriveInfo : PSDriveInfo
{
    private OdbcConnection connection;

    /// <summary>
    /// ODBC connection information.
    /// </summary>
    public OdbcConnection Connection
    {
        get { return connection; }
        set { connection = value; }
    }

    /// <summary>
    /// Constructor that takes one argument
    /// </summary>
    /// <param name="driveInfo">Drive provided by this provider</param>
    public AccessDBPSDriveInfo(PSDriveInfo driveInfo)
        : base(driveInfo)
    { }
}

```

```
    } // class AccessDBPSDriveInfo  
  
#endregion AccessDBPSDriveInfo  
}
```

See Also

[System.Management.Automation.Provider.ItemCmdletProvider](#)

[System.Management.Automation.Provider.ContainerCmdletProvider](#)

[System.Management.Automation.Provider.NavigationCmdletProvider](#)

[Designing Your Windows PowerShell Provider](#)

AccessDBProviderSample03

Article • 03/24/2025

This sample shows how to overwrite the `System.Management.Automation.Provider.ItemCmdletProvider.GetItem*` and `System.Management.Automation.Provider.ItemCmdletProvider.SetItem*` methods to support calls to the `Get-Item` and `Set-Item` cmdlets. The provider class in this sample derives from the `System.Management.Automation.Provider.ItemCmdletProvider` class.

Demonstrates

ⓘ Important

Your provider class will most likely derive from one of the following classes and possibly implement other provider interfaces:

- [System.Management.Automation.Provider.ItemCmdletProvider](#) class.
- [System.Management.Automation.Provider.ContainerCmdletProvider](#) class.
See [AccessDBProviderSample04](#).
- [System.Management.Automation.Provider.NavigationCmdletProvider](#) class.
See [AccessDBProviderSample05](#).

For more information about choosing which provider class to derive from based on provider features, see [Designing Your Windows PowerShell Provider](#).

This sample demonstrates the following:

- Declaring the `CmdletProvider` attribute.
- Defining a provider class that derives from the `System.Management.Automation.Provider.ItemCmdletProvider` class.
- Overwriting the `System.Management.Automation.Provider.DriveCmdletProvider.NewDrive*` method to change the behavior of the `New-PSDrive` cmdlet, allowing the user to create new drives. (This sample does not show how to add dynamic parameters to the `New-PSDrive` cmdlet.)
- Overwriting the `System.Management.Automation.Provider.DriveCmdletProvider.RemoveDrive*` method to support removing existing drives.

- Overwriting the [System.Management.Automation.Provider.ItemCmdletProvider.GetItem*](#) method to change the behavior of the `Get-Item` cmdlet, allowing the user to retrieve items from the data store. (This sample does not show how to add dynamic parameters to the `Get-Item` cmdlet.)
- Overwriting the [System.Management.Automation.Provider.ItemCmdletProvider.SetItem*](#) method to change the behavior of the `Set-Item` cmdlet, allowing the user to update the items in the data store. (This sample does not show how to add dynamic parameters to the `Get-Item` cmdlet.)
- Overwriting the [System.Management.Automation.Provider.ItemCmdletProvider.ItemExists*](#) method to change the behavior of the `Test-Path` cmdlet. (This sample does not show how to add dynamic parameters to the `Test-Path` cmdlet.)
- Overwriting the [System.Management.Automation.Provider.ItemCmdletProvider.IsValidPath*](#) method to determine if the provided path is valid.

Example

This sample shows how to overwrite the methods needed to get and set items in a Microsoft Access data base.

C#

```
using System;
using System.IO;
using System.Data;
using System.Data.Odbc;
using System.Collections.ObjectModel;
using System.Text;
using System.Diagnostics;
using System.Text.RegularExpressions;
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.ComponentModel;
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

    /// <summary>
    /// A PowerShell Provider which acts upon a access database.
    /// </summary>
    /// <remarks>
```

```

/// This example implements the item overloads.
/// </remarks>
[CmdletProvider("AccessDB", ProviderCapabilities.None)]


public class AccessDBProvider : ItemCmdletProvider
{
    #region Drive Manipulation

    /// <summary>
    /// Create a new drive. Create a connection to the database file and
    set
    /// the Connection property in the PSDriveInfo.
    /// </summary>
    /// <param name="drive">
    /// Information describing the drive to add.
    /// </param>
    /// <returns>The added drive.</returns>
    protected override PSDriveInfo NewDrive(PSDriveInfo drive)
    {
        // check if drive object is null
        if (drive == null)
        {
            WriteError(new ErrorRecord(
                new ArgumentNullException("drive"),
                "NullDrive",
                ErrorCategory.InvalidArgument,
                null)
            );
            return null;
        }

        // check if drive root is not null or empty
        // and if its an existing file
        if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root)
== false))
        {
            WriteError(new ErrorRecord(
                new ArgumentException("drive.Root"),
                "NoRoot",
                ErrorCategory.InvalidArgument,
                drive)
            );
            return null;
        }

        // create a new drive and create an ODBC connection to the new
        drive
        AccessDBPSDriveInfo accessDBPSDriveInfo = new
        AccessDBPSDriveInfo(drive);

        OdbcConnectionStringBuilder builder = new
        OdbcConnectionStringBuilder();

```

```

        builder.Driver = "Microsoft Access Driver (*.mdb)";
        builder.Add("DBQ", drive.Root);

        OdbcConnection conn = new
        OdbcConnection(builder.ConnectionString);
        conn.Open();
        accessDBPSDriveInfo.Connection = conn;

        return accessDBPSDriveInfo;
    } // NewDrive

    /// <summary>
    /// Removes a drive from the provider.
    /// </summary>
    /// <param name="drive">The drive to remove.</param>
    /// <returns>The drive removed.</returns>
    protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)
    {
        // check if drive object is null
        if (drive == null)
        {
            WriteError(new ErrorRecord(
                new ArgumentNullException("drive"),
                "NullDrive",
                ErrorCategory.InvalidArgument,
                drive)
            );
        }

        return null;
    }

    // close ODBC connection to the drive
    AccessDBPSDriveInfo accessDBPSDriveInfo = drive as
AccessDBPSDriveInfo;

    if (accessDBPSDriveInfo == null)
    {
        return null;
    }
    accessDBPSDriveInfo.Connection.Close();

    return accessDBPSDriveInfo;
} // RemoveDrive

#endregion Drive Manipulation

#region Item Methods

    /// <summary>
    /// Retrieves an item using the specified path.
    /// </summary>
    /// <param name="path">The path to the item to return.</param>
    protected override void GetItem(string path)
    {
        // check if the path represented is a drive

```

```

    if (PathIsDrive(path))
    {
        WriteItemObject(this.PSDriveInfo, path, true);
        return;
    }// if (PathIsDrive...

    // Get table name and row information from the path and do
    // necessary actions
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        DatabaseTableInfo table = GetTable(tableName);
        WriteItemObject(table, path, true);
    }
    else if (type == PathType.Row)
    {
        DatabaseRowInfo row = GetRow(tableName, rowNumber);
        WriteItemObject(row, path, false);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }

} // GetItem

/// <summary>
/// Set the content of a row of data specified by the supplied path
/// parameter.
/// </summary>
/// <param name="path">Specifies the path to the row whose columns
/// will be updated.</param>
/// <param name="values">Comma separated string of values</param>
protected override void SetItem(string path, object values)
{
    // Get type, table name and row number from the path specified
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type != PathType.Row)
    {
        WriteError(new ErrorRecord(new NotSupportedException(
            "SetNotSupported"), "", ErrorCategory.InvalidOperation, path));

        return;
    }
}

```

```

// Get in-memory representation of table
OdbcDataAdapter da = GetAdapterForTable(tableName);

if (da == null)
{
    return;
}
DataSet ds = GetDataSetForTable(da, tableName);
DataTable table = GetDataTable(ds, tableName);

if (rowNumber >= table.Rows.Count)
{
    // The specified row number has to be available. If not
    // NewItem has to be used to add a new row
    throw new ArgumentException("Row specified is not
available");
} // if (rowNum...

string[] colValues = (values as string).Split(',');
// set the specified row
DataRow row = table.Rows[rowNumber];

for (int i = 0; i < colValues.Length; i++)
{
    row[i] = colValues[i];
}

// Update the table
if (ShouldProcess(path, "SetItem"))
{
    da.Update(ds, tableName);
}

} // SetItem

/// <summary>
/// Test to see if the specified item exists.
/// </summary>
/// <param name="path">The path to the item to verify.</param>
/// <returns>True if the item is found.</returns>
protected override bool ItemExists(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        return true;
    }

    // Obtain type, table name and row number from path
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out

```

```

rowNumber);

DatabaseTableInfo table = GetTable(tableName);

if (type == PathType.Table)
{
    // if specified path represents a table then
DatabaseTableInfo
    // object for the same should exist
    if (table != null)
    {
        return true;
    }
}
else if (type == PathType.Row)
{
    // if specified path represents a row then DatabaseTableInfo
should
    // exist for the table and then specified row number must be
within
    // the maximum row count in the table
    if (table != null && rowNumber < table.RowCount)
    {
        return true;
    }
}

return false;

} // ItemExists

/// <summary>
/// Test to see if the specified path is syntactically valid.
/// </summary>
/// <param name="path">The path to validate.</param>
/// <returns>True if the specified path is valid.</returns>
protected override bool IsValidPath(string path)
{
    bool result = true;

    // check if the path is null or empty
    if (String.IsNullOrEmpty(path))
    {
        result = false;
    }

    // convert all separators in the path to a uniform one
    path = NormalizePath(path);

    // split the path into individual chunks
    string[] pathChunks = path.Split(pathSeparator.ToCharArray());

    foreach (string pathChunk in pathChunks)
    {
        if (pathChunk.Length == 0)

```

```

        {
            result = false;
        }
    }
    return result;
} // IsValidPath

#endregion Item Overloads

#region Helper Methods

/// <summary>
/// Checks if a given path is actually a drive name.
/// </summary>
/// <param name="path">The path to check.</param>
/// <returns>
/// True if the path given represents a drive, false otherwise.
/// </returns>
private bool PathIsDrive(string path)
{
    // Remove the drive name and first path separator. If the
    // path is reduced to nothing, it is a drive. Also if its
    // just a drive then there wont be any path separators
    if (String.IsNullOrEmpty(
        path.Replace(this.PSDriveInfo.Root, ""))
        String.IsNullOrEmpty(
            path.Replace(this.PSDriveInfo.Root + pathSeparator,
            "")))
    )

    {
        return true;
    }
    else
    {
        return false;
    }
} // PathIsDrive

/// <summary>
/// Breaks up the path into individual elements.
/// </summary>
/// <param name="path">The path to split.</param>
/// <returns>An array of path segments.</returns>
private string[] ChunkPath(string path)
{
    // Normalize the path before splitting
    string normalPath = NormalizePath(path);

    // Return the path with the drive name and first path
    // separator character removed, split by the path separator.
    string pathNoDrive = normalPath.Replace(this.PSDriveInfo.Root
        + pathSeparator, "");

    return pathNoDrive.Split(pathSeparator.ToCharArray());
}

```

```

} // ChunkPath

/// <summary>
/// Adapts the path, making sure the correct path separator
/// character is used.
/// </summary>
/// <param name="path"></param>
/// <returns></returns>
private string NormalizePath(string path)
{
    string result = path;

    if (!String.IsNullOrEmpty(path))
    {
        result = path.Replace("/", pathSeparator);
    }

    return result;
} // NormalizePath

/// <summary>
/// Chunks the path and returns the table name and the row number
/// from the path
/// </summary>
/// <param name="path">Path to chunk and obtain information</param>
/// <param name="tableName">Name of the table as represented in the
/// path</param>
/// <param name="rowNumber">Row number obtained from the path</param>
/// <returns>what the path represents</returns>
private PathType GetNamesFromPath(string path, out string tableName,
out int rowNumber)
{
    PathType retVal = PathType.Invalid;
    rowNumber = -1;
    tableName = null;

    // Check if the path specified is a drive
    if (PathIsDrive(path))
    {
        return PathType.Database;
    }

    // chunk the path into parts
    string[] pathChunks = ChunkPath(path);

    switch (pathChunks.Length)
    {
        case 1:
        {
            string name = pathChunks[0];

            if (TableNameIsValid(name))
            {
                tableName = name;
                retVal = PathType.Table;
            }
        }
    }
}

```

```

        }

    }

    break;

    case 2:
    {
        string name = pathChunks[0];

        if (TableNameIsValid(name))
        {
            tableName = name;
        }

        int number = SafeConvertRowNumber(pathChunks[1]);

        if (number >= 0)
        {
            rowNum = number;
            retVal = PathType.Row;
        }
        else
        {
            WriteError(new ErrorRecord(
                new ArgumentException("Row number is not
valid"),
                "RowNumberNotValid",
                ErrorCategory.InvalidArgument,
                path));
        }
    }
    break;

    default:
    {
        WriteError(new ErrorRecord(
            new ArgumentException("The path supplied has too
many segments"),
            "PathNotValid",
            ErrorCategory.InvalidArgument,
            path));
    }
    break;
} // switch(pathChunks...

return retVal;
} // GetNamesFromPath

/// <summary>
/// Throws an argument exception stating that the specified path does
/// not represent either a table or a row
/// </summary>
/// <param name="path">path which is invalid</param>
private void ThrowTerminatingInvalidPathException(string path)
{
    StringBuilder message = new StringBuilder("Path must represent

```

```

either a table or a row :");
message.Append(path);

throw new ArgumentException(message.ToString());
}

/// <summary>
/// Retrieve the list of tables from the database.
/// </summary>
/// <returns>
/// Collection of DatabaseTableInfo objects, each object representing
/// information about one database table
/// </returns>
private Collection<DatabaseTableInfo> GetTables()
{
    Collection<DatabaseTableInfo> results =
        new Collection<DatabaseTableInfo>();

    // using ODBC connection to the database and get the schema of
tables
    AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

    if (di == null)
    {
        return null;
    }

    OdbcConnection connection = di.Connection;
    DataTable dt = connection.GetSchema("Tables");
    int count;

    // iterate through all rows in the schema and create
DatabaseTableInfo
    // objects which represents a table
    foreach (DataRow dr in dt.Rows)
    {
        String tableName = dr["TABLE_NAME"] as String;
        DataColumnCollection columns = null;

        // find the number of rows in the table
        try
        {
            String cmd = "Select count(*) from \\" + tableName + "\\";
            OdbcCommand command = new OdbcCommand(cmd, connection);

            count = (Int32)command.ExecuteScalar();
        }
        catch
        {
            count = 0;
        }

        // create DatabaseTableInfo object representing the table
        DatabaseTableInfo table =
            new DatabaseTableInfo(dr, tableName, count, columns);
    }
}

```

```

        results.Add(table);
    } // foreach (DataRow...

    return results;
} // GetTables

/// <summary>
/// Return row information from a specified table.
/// </summary>
/// <param name="tableName">The name of the database table from
/// which to retrieve rows.</param>
/// <returns>Collection of row information objects.</returns>
private Collection<DatabaseRowInfo> GetRows(string tableName)
{
    Collection<DatabaseRowInfo> results =
        new Collection<DatabaseRowInfo>();

    // Obtain rows in the table and add it to the collection
    try
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);

        if (da == null)
        {
            return null;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        int i = 0;
        foreach (DataRow row in table.Rows)
        {
            results.Add(new DatabaseRowInfo(row,
i.ToString(CultureInfo.CurrentCulture)));
            i++;
        } // foreach (DataRow...
    }
    catch (Exception e)
    {
        WriteError(new ErrorRecord(e, "CannotAccessSpecifiedRows",
ErrorCategory.InvalidOperation, tableName));
    }
}

return results;
} // GetRows

/// <summary>
/// Retrieve information about a single table.
/// </summary>
/// <param name="tableName">The table for which to retrieve
/// data.</param>
/// <returns>Table information.</returns>

```

```

    private DatabaseTableInfo GetTable(string tableName)
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            if (String.Equals(tableName, table.Name,
StringComparison.OrdinalIgnoreCase))
            {
                return table;
            }
        }

        return null;
    } // GetTable

    /// <summary>
    /// Obtain a data adapter for the specified Table
    /// </summary>
    /// <param name="tableName">Name of the table to obtain the
    /// adapter for</param>
    /// <returns>Adapter object for the specified table</returns>
    /// <remarks>An adapter serves as a bridge between a DataSet (in
memory
    /// representation of table) and the data source</remarks>
    private OdbcDataAdapter GetAdapterForTable(string tableName)
    {
        OdbcDataAdapter da = null;
        AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

        if (di == null || !TableNameIsValid(tableName) ||
!TableIsPresent(tableName))
        {
            return null;
        }

        OdbcConnection connection = di.Connection;

        try
        {
            // Create a odbc data adpater. This can be sued to update the
            // data source with the records that will be created here
            // using data sets
            string sql = "Select * from " + tableName;
            da = new OdbcDataAdapter(new OdbcCommand(sql, connection));

            // Create a odbc command builder object. This will create sql
            // commands automatically for a single table, thus
            // eliminating the need to create new sql statements for
            // every operation to be done.
            OdbcCommandBuilder cmd = new OdbcCommandBuilder(da);

            // Open the connection if its not already open
            if (connection.State != ConnectionState.Open)
            {
                connection.Open();
            }
        }
    }
}

```

```

        }
        catch (Exception e)
        {
            WriteError(new ErrorRecord(e, "CannotAccessSpecifiedTable",
                ErrorCategory.InvalidOperation, tableName));
        }

        return da;
    } // GetAdapterForTable

    /// <summary>
    /// Gets the DataSet (in memory representation) for the table
    /// for the specified adapter
    /// </summary>
    /// <param name="adapter">Adapter to be used for obtaining
    /// the table</param>
    /// <param name="tableName">Name of the table for which a
    /// DataSet is required</param>
    /// <returns>The DataSet with the filled in schema</returns>
    private DataSet GetDataSetForTable(ODBCDataAdapter adapter, string
tableName)
{
    Debug.Assert(adapter != null);

    // Create a dataset object which will provide an in-memory
    // representation of the data being worked upon in the
    // data source.
    DataSet ds = new DataSet();

    // Create a table named "Table" which will contain the same
    // schema as in the data source.
    //adapter.FillSchema(ds, SchemaType.Source);
    adapter.Fill(ds, tableName);
    ds.Locale = CultureInfo.InvariantCulture;

    return ds;
} //GetDataSetForTable

    /// <summary>
    /// Get the DataTable object which can be used to operate on
    /// for the specified table in the data source
    /// </summary>
    /// <param name="ds">DataSet object which contains the tables
    /// schema</param>
    /// <param name="tableName">Name of the table</param>
    /// <returns>Corresponding DataTable object representing
    /// the table</returns>
    ///
    private DataTable GetDataTable(DataSet ds, string tableName)
{
    Debug.Assert(ds != null);
    Debug.Assert(tableName != null);

    DataTable table = ds.Tables[tableName];
    table.Locale = CultureInfo.InvariantCulture;
}

```

```

        return table;
    } // GetDataTable

    /// <summary>
    /// Retrieves a single row from the named table.
    /// </summary>
    /// <param name="tableName">The table that contains the
    /// numbered row.</param>
    /// <param name="row">The index of the row to return.</param>
    /// <returns>The specified table row.</returns>
    private DatabaseRowInfo GetRow(string tableName, int row)
    {
        Collection<DatabaseRowInfo> di = GetRows(tableName);

        // if the row is invalid write an appropriate error else return
        the
        // corresponding row information
        if (row < di.Count && row >= 0)
        {
            return di[row];
        }
        else
        {
            WriteError(new ErrorRecord(
                new ItemNotFoundException(),
                "RowNotFound",
                ErrorCategory.ObjectNotFound,
                row.ToString(CultureInfo.CurrentCulture))
            );
        }

        return null;
    } // GetRow

    /// <summary>
    /// Method to safely convert a string representation of a row number
    /// into its Int32 equivalent
    /// </summary>
    /// <param name="rowNumberAsStr">String representation of the row
    /// number</param>
    /// <remarks>If there is an exception, -1 is returned</remarks>
    private int SafeConvertRowNumber(string rowNumberAsStr)
    {
        int rowNumber = -1;
        try
        {
            rowNumber = Convert.ToInt32(rowNumberAsStr,
CultureInfo.CurrentCulture);
        }
        catch (FormatException fe)
        {
            WriteError(new ErrorRecord(fe, "RowStringFormatNotValid",
                ErrorCategory.InvalidData, rowNumberAsStr));
        }
    }
}

```

```

        catch (OverflowException oe)
        {
            WriteError(new ErrorRecord(oe,
"RowStringConversionToNumberFailed",
                ErrorCategory.InvalidData, rowNumberAsStr));
        }

        return rowNumber;
    } // SafeConvertRowNumber

    /// <summary>
    /// Check if a table name is valid
    /// </summary>
    /// <param name="tableName">Table name to validate</param>
    /// <remarks>Helps to check for SQL injection attacks</remarks>
    private bool TableNameIsValid(string tableName)
    {
        Regex exp = new Regex(pattern, RegexOptions.Compiled |
RegexOptions.IgnoreCase);

        if (exp.IsMatch(tableName))
        {
            return true;
        }
        WriteError(new ErrorRecord(
            new ArgumentException("Table name not valid"),
"TableNameNotValid",
                ErrorCategory.InvalidArgument, tableName));
        return false;
    } // TableNameIsValid

    /// <summary>
    /// Checks to see if the specified table is present in the
    /// database
    /// </summary>
    /// <param name="tableName">Name of the table to check</param>
    /// <returns>true, if table is present, false otherwise</returns>
    private bool TableIsPresent(string tableName)
    {
        // using ODBC connection to the database and get the schema of
tables
        AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;
        if (di == null)
        {
            return false;
        }

        OdbcConnection connection = di.Connection;
        DataTable dt = connection.GetSchema("Tables");

        // check if the specified tableName is available
        // in the list of tables present in the database
        foreach (DataRow dr in dt.Rows)
        {
            string name = dr["TABLE_NAME"] as string;

```

```

                if (name.Equals(tableName,
StringComparison.OrdinalIgnoreCase))
{
    return true;
}
}

WriteError(new ErrorRecord(
    new ArgumentException("Specified Table is not present in
database"), "TableNotAvailable",
    ErrorCategory.InvalidArgument, tableName));

return false;
}// TableIsPresent

#endregion Helper Methods

#region Private Properties

private string pathSeparator = "\\";
private static string pattern = @"^+[a-z]+[0-9]*_*$";

private enum PathType { Database, Table, Row, Invalid };

#endregion Private Properties
}

#endregion AccessDBProvider

#region Helper Classes

#region AccessDBPSDriveInfo

/// <summary>
/// Any state associated with the drive should be held here.
/// In this case, it's the connection to the database.
/// </summary>
internal class AccessDBPSDriveInfo : PSDriveInfo
{
    private OdbcConnection connection;

    /// <summary>
    /// ODBC connection information.
    /// </summary>
    public OdbcConnection Connection
    {
        get { return connection; }
        set { connection = value; }
    }

    /// <summary>
    /// Constructor that takes one argument
    /// </summary>
    /// <param name="driveInfo">Drive provided by this provider</param>
    public AccessDBPSDriveInfo(PSDriveInfo driveInfo)

```

```

        : base(driveInfo)
    }

} // class AccessDBPSDriveInfo

#endregion AccessDBPSDriveInfo

#region DatabaseTableInfo

/// <summary>
/// Contains information specific to the database table.
/// Similar to the DirectoryInfo class.
/// </summary>
public class DatabaseTableInfo
{
    /// <summary>
    /// Row from the "tables" schema
    /// </summary>
    public DataRow Data
    {
        get
        {
            return data;
        }
        set
        {
            data = value;
        }
    }
    private DataRow data;

    /// <summary>
    /// The table name.
    /// </summary>
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
    private String name;

    /// <summary>
    /// The number of rows in the table.
    /// </summary>
    public int RowCount
    {
        get
        {
            return rowCount;
        }
    }
}

```

```

        }
        set
        {
            rowCount = value;
        }
    }
    private int rowCount;

    /// <summary>
    /// The column definitions for the table.
    /// </summary>
    public DataColumnCollection Columns
    {
        get
        {
            return columns;
        }
        set
        {
            columns = value;
        }
    }
    private DataColumnCollection columns;

    /// <summary>
    /// Constructor.
    /// </summary>
    /// <param name="row">The row definition.</param>
    /// <param name="name">The table name.</param>
    /// <param name="rowCount">The number of rows in the table.</param>
    /// <param name="columns">Information on the column tables.</param>
    public DatabaseTableInfo(DataRow row, string name, int rowCount,
                             DataColumnCollection columns)
    {
        Name = name;
        Data = row;
        RowCount = rowCount;
        Columns = columns;
    } // DatabaseTableInfo
} // class DatabaseTableInfo

#endregion DatabaseTableInfo

#region DatabaseRowInfo

    /// <summary>
    /// Contains information specific to an individual table row.
    /// Analogous to the FileInfo class.
    /// </summary>
    public class DatabaseRowInfo
    {
        /// <summary>
        /// Row data information.
        /// </summary>
        public DataRow Data

```

```

{
    get
    {
        return data;
    }
    set
    {
        data = value;
    }
}
private DataRow data;

/// <summary>
/// The row index.
/// </summary>
public string RowNumber
{
    get
    {
        return rowNum;
    }
    set
    {
        rowNum = value;
    }
}
private string rowNum;

/// <summary>
/// Constructor.
/// </summary>
/// <param name="row">The row information.</param>
/// <param name="name">The row index.</param>
public DatabaseRowInfo(DataRow row, string name)
{
    RowNumber = name;
    Data = row;
} // DatabaseRowInfo
} // class DatabaseRowInfo

#endregion DatabaseRowInfo

#endregion Helper Classes
}

```

See Also

[System.Management.Automation.Provider.ItemCmdletProvider](#)

[System.Management.Automation.Provider.ContainerCmdletProvider](#)

[System.Management.Automation.Provider.NavigationCmdletProvider](#)

Designing Your Windows PowerShell Provider

AccessDBProviderSample04

Article • 03/24/2025

This sample shows how to overwrite container methods to support calls to the `Copy-Item`, `Get-ChildItem`, `New-Item`, and `Remove-Item` cmdlets. These methods should be implemented when the data store contains items that are containers. A container is a group of child items under a common parent item. The provider class in this sample derives from the [System.Management.Automation.Provider.ContainerCmdletProvider](#) class.

Demonstrates

Important

Your provider class will most likely derive from the
[System.Management.Automation.Provider.NavigationCmdletProvider](#)

This sample demonstrates the following:

- Declaring the `CmdletProvider` attribute.
- Defining a provider class that derives from the [System.Management.Automation.Provider.ContainerCmdletProvider](#) class.
- Overwriting the [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItem](#) method to change the behavior of the `Copy-Item` cmdlet which allows the user to copy items from one location to another. (This sample does not show how to add dynamic parameters to the `Copy-Item` cmdlet.)
- Overwriting the [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems](#) * method to change the behavior of the `Get-ChildItems` cmdlet, which allows the user to retrieve the child items of the parent item. (This sample does not show how to add dynamic parameters to the `Get-ChildItems` cmdlet.)
- Overwriting the [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildNames](#) \$* method to change the behavior of the `Get-ChildItems` cmdlet when the `Name` parameter of the cmdlet is specified.
- Overwriting the [System.Management.Automation.Provider.ContainerCmdletProvider.NewItem](#)*

method to change the behavior of the `New-Item` cmdlet, which allows the user to add items to the data store. (This sample does not show how to add dynamic parameters to the `New-Item` cmdlet.)

- Overwriting the `System.Management.Automation.Provider.ContainerCmdletProvider.RemoveItem*` method to change the behavior of the `Remove-Item` cmdlet. (This sample does not show how to add dynamic parameters to the `Remove-Item` cmdlet.)

Example

This sample shows how to overwrite the methods needed to copy, create, and remove items, as well as methods for getting the child items of a parent item.

C#

```
using System;
using System.IO;
using System.Data;
using System.Data.Odbc;
using System.Data.OleDb;
using System.Diagnostics;
using System.Collections.ObjectModel;
using System.Text;
using System.Text.RegularExpressions;
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.ComponentModel;
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

        /// <summary>
        /// A PowerShell Provider which acts upon an Access database
        /// </summary>
        /// <remarks>
        /// This example implements the container overloads</remarks>
        [CmdletProvider("AccessDB", ProviderCapabilities.None)]
        public class AccessDBProvider : ContainerCmdletProvider
    {

        #region Drive Manipulation

            /// <summary>
            /// Create a new drive. Create a connection to the database file and
            /// set
            /// the Connection property in the PSDriveInfo.
            /// </summary>
        
```

```

/// <param name="drive">
/// Information describing the drive to add.
/// </param>
/// <returns>The added drive.</returns>
protected override PSDriveInfo NewDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            null)
        );
    }

    return null;
}

// check if drive root is not null or empty
// and if its an existing file
if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root)
== false))
{
    WriteError(new ErrorRecord(
        new ArgumentException("drive.Root"),
        "NoRoot",
        ErrorCategory.InvalidArgument,
        drive)
    );
}

return null;
}

// create a new drive and create an ODBC connection to the new
drive
AccessDBPSDriveInfo accessDBPSDriveInfo = new
AccessDBPSDriveInfo(drive);

OdbcConnectionStringBuilder builder = new
OdbcConnectionStringBuilder();

builder.Driver = "Microsoft Access Driver (*.mdb)";
builder.Add("DBQ", drive.Root);

OdbcConnection conn = new
OdbcConnection(builder.ConnectionString);
conn.Open();
accessDBPSDriveInfo.Connection = conn;

return accessDBPSDriveInfo;
} // NewDrive

/// <summary>

```

```

/// Removes a drive from the provider.
/// </summary>
/// <param name="drive">The drive to remove.</param>
/// <returns>The drive removed.</returns>
protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            drive)
        );
    }

    return null;
}

// close ODBC connection to the drive
AccessDBPSDriveInfo accessDBPSDriveInfo = drive as
AccessDBPSDriveInfo;

if (accessDBPSDriveInfo == null)
{
    return null;
}
accessDBPSDriveInfo.Connection.Close();

return accessDBPSDriveInfo;
} // RemoveDrive

#endregion Drive Manipulation

#region Item Methods

/// <summary>
/// Retrieves an item using the specified path.
/// </summary>
/// <param name="path">The path to the item to return.</param>
protected override void GetItem(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        WriteItemObject(this.PSDriveInfo, path, true);
        return;
    } // if (PathIsDrive...

    // Get table name and row information from the path and do
    // necessary actions
    string tableName;
    int rowCount;

    PathType type = GetNamesFromPath(path, out tableName, out

```

```

rowNumber);

    if (type == PathType.Table)
    {
        DatabaseTableInfo table = GetTable(tableName);
        WriteItemObject(table, path, true);
    }
    else if (type == PathType.Row)
    {
        DatabaseRowInfo row = GetRow(tableName, rowNumber);
        WriteItemObject(row, path, false);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }

} // GetItem

/// <summary>
/// Set the content of a row of data specified by the supplied path
/// parameter.
/// </summary>
/// <param name="path">Specifies the path to the row whose columns
/// will be updated.</param>
/// <param name="values">Comma separated string of values</param>
protected override void SetItem(string path, object values)
{
    // Get type, table name and row number from the path specified
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type != PathType.Row)
    {
        WriteError(new ErrorRecord(new NotSupportedException(
            "SetNotSupported"), "",  

            ErrorCategory.InvalidOperation, path));

        return;
    }

    // Get in-memory representation of table
    OdbcDataAdapter da = GetAdapterForTable(tableName);

    if (da == null)
    {
        return;
    }
    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    if (rowNumber >= table.Rows.Count)

```

```

    {
        // The specified row number has to be available. If not
        // NewItem has to be used to add a new row
        throw new ArgumentException("Row specified is not
available");
    } // if (rowNum...

    string[] colValues = (values as string).Split(',');
}

// set the specified row
DataRow row = table.Rows[rowNumber];

for (int i = 0; i < colValues.Length; i++)
{
    row[i] = colValues[i];
}

// Update the table
if (ShouldProcess(path, "SetItem"))
{
    da.Update(ds, tableName);
}

} // SetItem

/// <summary>
/// Test to see if the specified item exists.
/// </summary>
/// <param name="path">The path to the item to verify.</param>
/// <returns>True if the item is found.</returns>
protected override bool ItemExists(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        return true;
    }

    // Obtain type, table name and row number from path
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    DatabaseTableInfo table = GetTable(tableName);

    if (type == PathType.Table)
    {
        // if specified path represents a table then
DatabaseTableInfo
            // object for the same should exist
            if (table != null)
            {
                return true;
            }
        }
    }
}

```

```

        }
    }
    else if (type == PathType.Row)
    {
        // if specified path represents a row then DatabaseTableInfo
should
        // exist for the table and then specified row number must be
within
        // the maximum row count in the table
        if (table != null && rowNumber < table.RowCount)
        {
            return true;
        }
    }

    return false;
}

} // ItemExists

/// <summary>
/// Test to see if the specified path is syntactically valid.
/// </summary>
/// <param name="path">The path to validate.</param>
/// <returns>True if the specified path is valid.</returns>
protected override bool IsValidPath(string path)
{
    bool result = true;

    // check if the path is null or empty
    if (String.IsNullOrEmpty(path))
    {
        result = false;
    }

    // convert all separators in the path to a uniform one
    path = NormalizePath(path);

    // split the path into individual chunks
    string[] pathChunks = path.Split(pathSeparator.ToCharArray());

    foreach (string pathChunk in pathChunks)
    {
        if (pathChunk.Length == 0)
        {
            result = false;
        }
    }
    return result;
} // IsValidPath

#endregion Item Overloads

#region Container Overloads

/// <summary>

```

```

/// Return either the tables in the database or the datarows
/// </summary>
/// <param name="path">The path to the parent</param>
/// <param name="recurse">True to return all child items recursively.
/// </param>
protected override void GetChildItems(string path, bool recurse)
{
    // If path represented is a drive then the children in the path
are
    // tables. Hence all tables in the drive represented will have to
be
    // returned
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table, path, true);

            // if the specified item exists and recurse has been set
then
            // all child items within it have to be obtained as well
            if (ItemExists(path) && recurse)
            {
                GetChildItems(path + pathSeparator + table.Name,
recurse);
            }
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
else
{
    // Get the table name, row number and type of path from the
    // path specified
    string tableName;
    int rowCount;

    PathType type = GetNamesFromPath(path, out tableName, out
rowCount);

    if (type == PathType.Table)
    {
        // Obtain all the rows within the table
        foreach (DatabaseRowInfo row in GetRows(tableName))
        {
            WriteItemObject(row, path + pathSeparator +
row.RowNumber,
                            false);
        } // foreach (DatabaseRowInfo...
    }
    else if (type == PathType.Row)
    {
        // In this case the user has directly specified a row,
hence
        // just give that particular row
        DatabaseRowInfo row = GetRow(tableName, rowCount);
        WriteItemObject(row, path + pathSeparator +

```

```

row.RowNumber,
                                false);
}
else
{
    // In this case, the path specified is not valid
    ThrowTerminatingInvalidPathException(path);
}
} // else
} // GetChildItems

/// <summary>
/// Return the names of all child items.
/// </summary>
/// <param name="path">The root path.</param>
/// <param name="returnContainers">Not used.</param>
protected override void GetChildNames(string path,
                                      ReturnContainers returnContainers)
{
    // If the path represented is a drive, then the child items are
    // tables. get the names of all the tables in the drive.
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table.Name, path, true);
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get type, table name and row number from path specified
        string tableName;
        int rowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Table)
        {
            // Get all the rows in the table and then write out the
            // row numbers.
            foreach (DatabaseRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row.RowNumber, path, false);
            } // foreach (DatabaseRowInfo...
        }
        else if (type == PathType.Row)
        {
            // In this case the user has directly specified a row,
            hence
            // just give that particular row
            DatabaseRowInfo row = GetRow(tableName, rowNumber);

            WriteItemObject(row.RowNumber, path, false);
        }
    }
}

```

```

        else
        {
            ThrowTerminatingInvalidOperationException(path);
        }
    } // else
} // GetChildNames

/// <summary>
/// Determines if the specified path has child items.
/// </summary>
/// <param name="path">The path to examine.</param>
/// <returns>
/// True if the specified path has child items.
/// </returns>
protected override bool HasChildItems(string path)
{
    if (PathIsDrive(path))
    {
        return true;
    }

    return (ChunkPath(path).Length == 1);
} // HasChildItems

/// <summary>
/// Creates a new item at the specified path.
/// </summary>
///
/// <param name="path">
/// The path to the new item.
/// </param>
///
/// <param name="type">
/// Type for the object to create. "Table" for creating a new table
and
/// "Row" for creating a new row in a table.
/// </param>
///
/// <param name="newValue">
/// Object for creating new instance of a type at the specified path.
For
/// creating a "Table" the object parameter is ignored and for
creating
/// a "Row" the object must be of type string which will contain
comma
/// separated values of the rows to insert.
/// </param>
protected override void NewItem(string path, string type,
                                object newValue)
{
    string tableName;
    int rowNumber;

    PathType pt = GetNamesFromPath(path, out tableName, out
rowNumber);
}

```

```

    if (pt == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }

    // Check if type is either "table" or "row", if not throw an
    // exception
    if (!String.Equals(type, "table",
 StringComparison.OrdinalIgnoreCase)
        && !String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
    {
        WriteError(new ErrorRecord
            (new ArgumentException("Type must be either
a table or row"),
             "CannotCreateSpecifiedObject",
             ErrorCategory.InvalidArgument,
             path
            )
        );
    }

    throw new ArgumentException("This provider can only create
items of type \"table\" or \"row\"");
}

// Path type is the type of path of the container. So if a drive
// is specified, then a table can be created under it and if a
table
of
by
row
based
// is specified, then a row can be created under it. For the sake
// completeness, if a row is specified, then if the row specified
// the path does not exist, a new row is created. However, the
// number may not match as the row numbers only get incremented
// on the number of rows

if (PathIsDrive(path))
{
    if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
    {
        // Execute command using ODBC connection to create a
table
        try
        {
            // create the table using an sql statement
            string newTableName = newItemValue.ToString();

            if (!TableNameIsValid(newTableName))
            {
                return;
            }
        }
    }
}

```

```

        string sql = "create table " + newTableName
                    + " (ID INT)";

        // Create the table using the Odbc connection from
        the
        // drive.
        AccessDBPSDriveInfo di = this.PSDriveInfo as
        AccessDBPSDriveInfo;

        if (di == null)
        {
            return;
        }
        OdbcConnection connection = di.Connection;

        if (ShouldProcess(newTableName, "create"))
        {
            OdbcCommand cmd = new OdbcCommand(sql,
connection);
            cmd.ExecuteScalar();
        }
        catch (Exception ex)
        {
            WriteError(new ErrorRecord(ex,
"CannotCreateSpecifiedTable",
ErrorCategory.InvalidOperation, path)
);
        }
    } // if (String...
    else if (String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
    {
        throw new
            ArgumentException("A row cannot be created under a
database, specify a path that represents a Table");
    }
} // if (PathIsDrive...
else
{
    if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
    {
        if (rowNumber < 0)
        {
            throw new
                ArgumentException("A table cannot be created
within another table, specify a path that represents a database");
        }
        else
        {
            throw new
                ArgumentException("A table cannot be created
inside a row, specify a path that represents a database");
        }
    }
}

```

```

        } //if (String.Equals....  

        // if path specified is a row, create a new row  

        else if (String.Equals(type, "row",  

StringComparison.OrdinalIgnoreCase))  

        {  

            // The user is required to specify the values to be  

inserted  

            // into the table in a single string separated by commas  

            string value = newItemValue as string;  

            if (String.IsNullOrEmpty(value))  

            {  

                throw new  

                    ArgumentException("Value argument must have comma  

separated values of each column in a row");  

            }
            string[] rowValues = value.Split(',');
  

            OdbcDataAdapter da = GetAdapterForTable(tableName);  

            if (da == null)
            {
                return;
            }
  

            DataSet ds = GetDataSetForTable(da, tableName);
            DataTable table = GetDataTable(ds, tableName);
  

            if (rowValues.Length != table.Columns.Count)
            {
                string message =
                    String.Format(CultureInfo.CurrentCulture,
                        "The table has {0} columns and  

the value specified must have so many comma separated values",
                        table.Columns.Count);
  

                throw new ArgumentException(message);
            }
  

            if (!Force && (rowNumber >=0 && rowNumber <
table.Rows.Count))
            {
                string message =
String.Format(CultureInfo.CurrentCulture,
                    "The row {0} already  

exists. To create a new row specify row number as {1}, or specify path to a  

table, or use the -Force parameter",
                    rowNumber,
table.Rows.Count);
  

                throw new ArgumentException(message);
            }
  

            if (rowNumber > table.Rows.Count)
            {

```

```

                string message =
String.Format(CultureInfo.CurrentCulture,
                           "To create a new row specify row
number as {0}, or specify path to a table",
                           table.Rows.Count);

                throw new ArgumentException(message);
            }

// Create a new row and update the row with the input
// provided by the user
DataRow row = table.NewRow();
for (int i = 0; i < rowValues.Length; i++)
{
    row[i] = rowValues[i];
}
table.Rows.Add(row);

if (ShouldProcess(tableName, "update rows"))
{
    // Update the table from memory back to the data
source
    da.Update(ds, tableName);
}

} // else if (String...
} // else ...

} // NewItem

/// <summary>
/// Copies an item at the specified path to the location specified
/// </summary>
///
/// <param name="path">
/// Path of the item to copy
/// </param>
///
/// <param name="copyPath">
/// Path of the item to copy to
/// </param>
///
/// <param name="recurse">
/// Tells the provider to recurse subcontainers when copying
/// </param>
///
protected override void CopyItem(string path, string copyPath, bool
recurse)
{
    string tableName, copyTableName;
    int rowCount, copyRowCount;

    PathType type = GetNamesFromPath(path, out tableName, out
rowCount);
    PathType copyType = GetNamesFromPath(copyPath, out copyTableName,

```

```

    out copyRowNumber);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(copyPath);
    }

    // Get the table and the table to copy to
    OdbcDataAdapter da = GetAdapterForTable(tableName);
    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    OdbcDataAdapter cda = GetAdapterForTable(copyTableName);
    if (cda == null)
    {
        return;
    }

    DataSet cds = GetDataSetForTable(cda, copyTableName);
    DataTable copyTable = GetDataTable(cds, copyTableName);

    // if source represents a table
    if (type == PathType.Table)
    {
        // if copyPath does not represent a table
        if (copyType != PathType.Table)
        {
            ArgumentException e = new ArgumentException("Table can
only be copied on to another table location");

            WriteError(new ErrorRecord(e, "PathNotValid",
                ErrorCategory.InvalidArgument, copyPath));

            throw e;
        }

        // if table already exists then force parameter should be set
        // to force a copy
        if (!Force && GetTable(copyTableName) != null)
        {
            throw new ArgumentException("Specified path already
exists");
        }

        for (int i = 0; i < table.Rows.Count; i++)
    }
}

```

```

    {
        DataRow row = table.Rows[i];
        DataRow copyRow = copyTable.NewRow();

        copyRow.ItemArray = row.ItemArray;
        copyTable.Rows.Add(copyRow);
    }
} // if (type == ...
// if source represents a row
else
{
    if (copyType == PathType.Row)
    {
        if (!Force && (copyRowNumber < copyTable.Rows.Count))
        {
            throw new ArgumentException("Specified path already
exists.");
        }

        DataRow row = table.Rows[rowNumber];
        DataRow copyRow = null;

        if (copyRowNumber < copyTable.Rows.Count)
        {
            // copy to an existing row
            copyRow = copyTable.Rows[copyRowNumber];
            copyRow.ItemArray = row.ItemArray;
            copyRow[0] = GetNextID(copyTable);
        }
        else if (copyRowNumber == copyTable.Rows.Count)
        {
            // copy to the next row in the table that will
            // be created
            copyRow = copyTable.NewRow();
            copyRow.ItemArray = row.ItemArray;
            copyRow[0] = GetNextID(copyTable);
            copyTable.Rows.Add(copyRow);
        }
        else
        {
            // attempting to copy to a nonexistent row or a row
            // that cannot be created now - throw an exception
            string message =
String.Format(CultureInfo.CurrentCulture,
                         "The item cannot be specified
to the copied row. Specify row number as {0}, or specify a path to the
table.",

table.Rows.Count);

            throw new ArgumentException(message);
        }
    }
    else
    {
        // destination path specified represents a table,

```

```

        // create a new row and copy the item
        DataRow copyRow = copyTable.NewRow();
        copyRow.ItemArray = table.Rows[rowNumber].ItemArray;
        copyRow[0] = GetNextID(copyTable);
        copyTable.Rows.Add(copyRow);
    }

}

if (ShouldProcess(copyTableName, "CopyItems"))
{
    cda.Update(cds, copyTableName);
}

} //CopyItem

/// <summary>
/// Removes (deletes) the item at the specified path
/// </summary>
///
/// <param name="path">
/// The path to the item to remove.
/// </param>
///
/// <param name="recurse">
/// True if all children in a subtree should be removed, false if
only
    /// the item at the specified path should be removed. Is applicable
    /// only for container (table) items. Its ignored otherwise (even if
    /// specified).
    /// </param>
    ///
    /// <remarks>
    /// There are no elements in this store which are hidden from the
user.
    /// Hence this method will not check for the presence of the Force
    /// parameter
    /// </remarks>
    ///
protected override void RemoveItem(string path, bool recurse)
{
    string tableName;
    int rowNumber = 0;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        // if recurse flag has been specified, delete all the rows as
well
        if (recurse)
        {
            OdbcDataAdapter da = GetAdapterForTable(tableName);
            if (da == null)
            {

```

```

        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    for (int i = 0; i < table.Rows.Count; i++)
    {
        table.Rows[i].Delete();
    }

    if (ShouldProcess(path, "RemoveItem"))
    {
        da.Update(ds, tableName);
        RemoveTable(tableName);
    }
    } //if (recurse...
    else
    {
        // Remove the table
        if (ShouldProcess(path, "RemoveItem"))
        {
            RemoveTable(tableName);
        }
    }
}
else if (type == PathType.Row)
{
    OdbcDataAdapter da = GetAdapterForTable(tableName);
    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    table.Rows[rowNumber].Delete();

    if (ShouldProcess(path, "RemoveItem"))
    {
        da.Update(ds, tableName);
    }
}
else
{
    ThrowTerminatingInvalidOperationException(path);
}

} // RemoveItem

#endregion Container Overloads

#region Helper Methods

```

```

/// <summary>
/// Checks if a given path is actually a drive name.
/// </summary>
/// <param name="path">The path to check.</param>
/// <returns>
/// True if the path given represents a drive, false otherwise.
/// </returns>
private bool PathIsDrive(string path)
{
    // Remove the drive name and first path separator. If the
    // path is reduced to nothing, it is a drive. Also if its
    // just a drive then there wont be any path separators
    if (String.IsNullOrEmpty(
        path.Replace(this.PSDriveInfo.Root, "")) ||
        String.IsNullOrEmpty(
        path.Replace(this.PSDriveInfo.Root + pathSeparator,
        "")))
    {
        return true;
    }
    else
    {
        return false;
    }
} // PathIsDrive

/// <summary>
/// Breaks up the path into individual elements.
/// </summary>
/// <param name="path">The path to split.</param>
/// <returns>An array of path segments.</returns>
private string[] ChunkPath(string path)
{
    // Normalize the path before splitting
    string normalPath = NormalizePath(path);

    // Return the path with the drive name and first path
    // separator character removed, split by the path separator.
    string pathNoDrive = normalPath.Replace(this.PSDriveInfo.Root
        + pathSeparator, "");

    return pathNoDrive.Split(pathSeparator.ToCharArray());
} // ChunkPath

/// <summary>
/// Adapts the path, making sure the correct path separator
/// character is used.
/// </summary>
/// <param name="path"></param>
/// <returns></returns>
private string NormalizePath(string path)
{
    string result = path;

```

```

    if (!String.IsNullOrEmpty(path))
    {
        result = path.Replace("/", pathSeparator);
    }

    return result;
} // NormalizePath

/// <summary>
/// Chunks the path and returns the table name and the row number
/// from the path
/// </summary>
/// <param name="path">Path to chunk and obtain information</param>
/// <param name="tableName">Name of the table as represented in the
/// path</param>
/// <param name="rowNumber">Row number obtained from the path</param>
/// <returns>what the path represents</returns>
private PathType GetNamesFromPath(string path, out string tableName,
out int rowNumber)
{
    PathType retVal = PathType.Invalid;
    rowNumber = -1;
    tableName = null;

    // Check if the path specified is a drive
    if (PathIsDrive(path))
    {
        return PathType.Database;
    }

    // chunk the path into parts
    string[] pathChunks = ChunkPath(path);

    switch (pathChunks.Length)
    {
        case 1:
        {
            string name = pathChunks[0];

            if (TableNameIsValid(name))
            {
                tableName = name;
                retVal = PathType.Table;
            }
        }
        break;

        case 2:
        {
            string name = pathChunks[0];

            if (TableNameIsValid(name))
            {
                tableName = name;
            }
        }
    }
}

```

```

        }

        int number = SafeConvertRowNumber(pathChunks[1]);

        if (number >= 0)
        {
            rowNum = number;
            retVal = PathType.Row;
        }
        else
        {
            WriteError(new ErrorRecord(
                new ArgumentException("Row number is not
valid"),
                "RowNumberNotValid",
                ErrorCategory.InvalidArgument,
                path));
        }
    }
    break;

default:
{
    WriteError(new ErrorRecord(
        new ArgumentException("The path supplied has too
many segments"),
        "PathNotValid",
        ErrorCategory.InvalidArgument,
        path));
}
break;
} // switch(pathChunks...

return retVal;
} // GetNamesFromPath

/// <summary>
/// Throws an argument exception stating that the specified path does
/// not represent either a table or a row
/// </summary>
/// <param name="path">path which is invalid</param>
private void ThrowTerminatingInvalidPathException(string path)
{
    StringBuilder message = new StringBuilder("Path must represent
either a table or a row :");
    message.Append(path);

    throw new ArgumentException(message.ToString());
}

/// <summary>
/// Retrieve the list of tables from the database.
/// </summary>
/// <returns>
/// Collection of DatabaseTableInfo objects, each object representing

```

```

    /// information about one database table
    /// </returns>
    private Collection<DatabaseTableInfo> GetTables()
    {
        Collection<DatabaseTableInfo> results =
            new Collection<DatabaseTableInfo>();

        // using ODBC connection to the database and get the schema of
        tables
        AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

        if (di == null)
        {
            return null;
        }

        OdbcConnection connection = di.Connection;
        DataTable dt = connection.GetSchema("Tables");
        int count;

        // iterate through all rows in the schema and create
        DatabaseTableInfo
        // objects which represents a table
        foreach (DataRow dr in dt.Rows)
        {
            String tableName = dr["TABLE_NAME"] as String;
            DataColumnCollection columns = null;

            // find the number of rows in the table
            try
            {
                String cmd = "Select count(*) from \\" + tableName +
                "\\";
                OdbcCommand command = new OdbcCommand(cmd, connection);

                count = (Int32)command.ExecuteScalar();
            }
            catch
            {
                count = 0;
            }

            // create DatabaseTableInfo object representing the table
            DatabaseTableInfo table =
                new DatabaseTableInfo(dr, tableName, count, columns);

            results.Add(table);
        } // foreach (DataRow...

        return results;
    } // GetTables

    /// <summary>
    /// Return row information from a specified table.
    /// </summary>

```

```

/// <param name="tableName">The name of the database table from
/// which to retrieve rows.</param>
/// <returns>Collection of row information objects.</returns>
private Collection<DatabaseRowInfo> GetRows(string tableName)
{
    Collection<DatabaseRowInfo> results =
        new Collection<DatabaseRowInfo>();

    // Obtain rows in the table and add it to the collection
    try
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);

        if (da == null)
        {
            return null;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        int i = 0;
        foreach (DataRow row in table.Rows)
        {
            results.Add(new DatabaseRowInfo(row,
i.ToString(CultureInfo.CurrentCulture)));
            i++;
        } // foreach (DataRow...
    }
    catch (Exception e)
    {
        WriteError(new ErrorRecord(e, "CannotAccessSpecifiedRows",
            ErrorCategory.InvalidOperation, tableName));
    }

    return results;
} // GetRows

/// <summary>
/// Retrieve information about a single table.
/// </summary>
/// <param name="tableName">The table for which to retrieve
/// data.</param>
/// <returns>Table information.</returns>
private DatabaseTableInfo GetTable(string tableName)
{
    foreach (DatabaseTableInfo table in GetTables())
    {
        if (String.Equals(tableName, table.Name,
StringComparison.OrdinalIgnoreCase))
        {
            return table;
        }
    }
}

```

```

        return null;
    } // GetTable

    /// <summary>
    /// Removes the specified table from the database
    /// </summary>
    /// <param name="tableName">Name of the table to remove</param>
    private void RemoveTable(string tableName)
    {
        // validate if tablename is valid and if table is present
        if (String.IsNullOrEmpty(tableName) ||
!TableNameIsValid(tableName) || !TableIsPresent(tableName))
        {
            return;
        }

        // Execute command using ODBC connection to remove a table
        try
        {
            // delete the table using an sql statement
            string sql = "drop table " + tableName;

            AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

            if (di == null)
            {
                return;
            }
            OdbcConnection connection = di.Connection;

            OdbcCommand cmd = new OdbcCommand(sql, connection);
            cmd.ExecuteScalar();
        }
        catch (Exception ex)
        {
            WriteError(new ErrorRecord(ex, "CannotRemoveSpecifiedTable",
ErrorCategory.InvalidOperation, null));
        }
    }

} // RemoveTable

    /// <summary>
    /// Obtain a data adapter for the specified Table
    /// </summary>
    /// <param name="tableName">Name of the table to obtain the
    /// adapter for</param>
    /// <returns>Adapter object for the specified table</returns>
    /// <remarks>An adapter serves as a bridge between a DataSet (in
memory
    /// representation of table) and the data source</remarks>
    private OdbcDataAdapter GetAdapterForTable(string tableName)
    {

```

```

OdbcDataAdapter da = null;
AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

if (di == null || !TableNameIsValid(tableName)
|| !TableIsPresent(tableName))
{
    return null;
}

OdbcConnection connection = di.Connection;

try
{
    // Create a odbc data adpater. This can be sued to update the
    // data source with the records that will be created here
    // using data sets
    string sql = "Select * from " + tableName;
    da = new OdbcDataAdapter(new OdbcCommand(sql, connection));

    // Create a odbc command builder object. This will create sql
    // commands automatically for a single table, thus
    // eliminating the need to create new sql statements for
    // every operation to be done.
    OdbcCommandBuilder cmd = new OdbcCommandBuilder(da);

    // Set the delete cmd for the table here
    sql = "Delete from " + tableName + " where ID = ?";
    da.DeleteCommand = new OdbcCommand(sql, connection);

    // Specify a DeleteCommand parameter based on the "ID"
    // column
    da.DeleteCommand.Parameters.Add(new OdbcParameter());
    da.DeleteCommand.Parameters[0].SourceColumn = "ID";

    // Create an InsertCommand based on the sql string
    // Insert into "tablename" values (?,?,?) where
    // ? represents a column in the table. Note that
    // the number of ? will be equal to the number of
    // columnds
    DataSet ds = new DataSet();

    da.FillSchema(ds, SchemaType.Source);
    ds.Locale = CultureInfo.InvariantCulture;

    sql = "Insert into " + tableName + " values ( ";
    for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
    {
        sql += "?, ";
    }
    sql = sql.Substring(0, sql.Length - 2);
    sql += ")";
    da.InsertCommand = new OdbcCommand(sql, connection);

    // Create parameters for the InsertCommand based on the
    // captions of each column
}

```

```

        for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
        {
            da.InsertCommand.Parameters.Add(new OdbcParameter());
            da.InsertCommand.Parameters[i].SourceColumn =
                ds.Tables["Table"].Columns[i].Caption;

        }

        // Open the connection if its not already open
        if (connection.State != ConnectionState.Open)
        {
            connection.Open();
        }
    }
    catch (Exception e)
    {
        WriteError(new ErrorRecord(e, "CannotAccessSpecifiedTable",
            ErrorCategory.InvalidOperation, tableName));
    }

    return da;
} // GetAdapterForTable

/// <summary>
/// Gets the DataSet (in memory representation) for the table
/// for the specified adapter
/// </summary>
/// <param name="adapter">Adapter to be used for obtaining
/// the table</param>
/// <param name="tableName">Name of the table for which a
/// DataSet is required</param>
/// <returns>The DataSet with the filled in schema</returns>
private DataSet GetDataSetForTable(OdbcDataAdapter adapter, string
tableName)
{
    Debug.Assert(adapter != null);

    // Create a dataset object which will provide an in-memory
    // representation of the data being worked upon in the
    // data source.
    DataSet ds = new DataSet();

    // Create a table named "Table" which will contain the same
    // schema as in the data source.
    //adapter.FillSchema(ds, SchemaType.Source);
    adapter.Fill(ds, tableName);
    ds.Locale = CultureInfo.InvariantCulture;

    return ds;
} //GetDataSetForTable

/// <summary>
/// Get the DataTable object which can be used to operate on
/// for the specified table in the data source
/// </summary>

```

```

/// <param name="ds">DataSet object which contains the tables
/// schema</param>
/// <param name="tableName">Name of the table</param>
/// <returns>Corresponding DataTable object representing
/// the table</returns>
///
private DataTable GetDataTable(DataSet ds, string tableName)
{
    Debug.Assert(ds != null);
    Debug.Assert(tableName != null);

    DataTable table = ds.Tables[tableName];
    table.Locale = CultureInfo.InvariantCulture;

    return table;
} // GetDataTable

/// <summary>
/// Retrieves a single row from the named table.
/// </summary>
/// <param name="tableName">The table that contains the
/// numbered row.</param>
/// <param name="row">The index of the row to return.</param>
/// <returns>The specified table row.</returns>
private DatabaseRowInfo GetRow(string tableName, int row)
{
    Collection<DatabaseRowInfo> di = GetRows(tableName);

    // if the row is invalid write an appropriate error else return
the
    // corresponding row information
    if (row < di.Count && row >= 0)
    {
        return di[row];
    }
    else
    {
        WriteError(new ErrorRecord(
            new ItemNotFoundException(),
            "RowNotFound",
            ErrorCategory.ObjectNotFound,
            row.ToString(CultureInfo.CurrentCulture)));
    }

    return null;
} // GetRow

/// <summary>
/// Method to safely convert a string representation of a row number
/// into its Int32 equivalent
/// </summary>
/// <param name="rowNumberAsStr">String representation of the row
/// number</param>
/// <remarks>If there is an exception, -1 is returned</remarks>

```

```

private int SafeConvertRowNumber(string rowNumberAsStr)
{
    int rowNumber = -1;
    try
    {
        rowNumber = Convert.ToInt32(rowNumberAsStr,
CultureInfo.CurrentCulture);
    }
    catch (FormatException fe)
    {
        WriteError(new ErrorRecord(fe, "RowStringFormatNotValid",
ErrorCategory.InvalidData, rowNumberAsStr));
    }
    catch (OverflowException oe)
    {
        WriteError(new ErrorRecord(oe,
"RowStringConversionToNumberFailed",
ErrorCategory.InvalidData, rowNumberAsStr));
    }

    return rowNumber;
} // SafeConvertRowNumber

/// <summary>
/// Check if a table name is valid
/// </summary>
/// <param name="tableName">Table name to validate</param>
/// <remarks>Helps to check for SQL injection attacks</remarks>
private bool TableNameIsValid(string tableName)
{
    Regex exp = new Regex(pattern, RegexOptions.Compiled |
RegexOptions.IgnoreCase);

    if (exp.IsMatch(tableName))
    {
        return true;
    }
    WriteError(new ErrorRecord(
        new ArgumentException("Table name not valid"),
"TableNameNotValid",
ErrorCategory.InvalidArgument, tableName));
    return false;
} // TableNameIsValid

/// <summary>
/// Checks to see if the specified table is present in the
/// database
/// </summary>
/// <param name="tableName">Name of the table to check</param>
/// <returns>true, if table is present, false otherwise</returns>
private bool TableIsPresent(string tableName)
{
    // using ODBC connection to the database and get the schema of
tables
    AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;
}

```

```

        if (di == null)
    {
        return false;
    }

    OdbcConnection connection = di.Connection;
    DataTable dt = connection.GetSchema("Tables");

    // check if the specified tableName is available
    // in the list of tables present in the database
    foreach (DataRow dr in dt.Rows)
    {
        string name = dr["TABLE_NAME"] as string;
        if (name.Equals(tableName,
StringComparison.OrdinalIgnoreCase))
        {
            return true;
        }
    }

    WriteError(new ErrorRecord(
        new ArgumentException("Specified Table is not present in
database"), "TableNotAvailable",
        ErrorCategory.InvalidArgument, tableName));

    return false;
}// TableIsPresent

/// <summary>
/// Gets the next available ID in the table
/// </summary>
/// <param name="table">DataTable object representing the table to
/// search for ID</param>
/// <returns>next available id</returns>
private int GetNextID(DataTable table)
{
    int big = 0;
    int id = 0;

    for (int i = 0; i < table.Rows.Count; i++)
    {
        DataRow row = table.Rows[i];

        object o = row["ID"];

        if (o.GetType().Name.Equals("Int16"))
        {
            id = (int)(short)o;
        }
        else
        {
            id = (int)o;
        }

        if (big < id)

```

```

        {
            big = id;
        }
    }

    big++;
    return big;
}

#endregion Helper Methods

#region Private Properties

private string pathSeparator = "\\";
private static string pattern = @"^+[a-z]+[0-9]*_*$";

private enum PathType { Database, Table, Row, Invalid };

#endregion Private Properties
}

#endregion AccessDBProvider

#region Helper Classes

#region AccessDBPSDriveInfo

/// <summary>
/// Any state associated with the drive should be held here.
/// In this case, it's the connection to the database.
/// </summary>
internal class AccessDBPSDriveInfo : PSDriveInfo
{
    private OdbcConnection connection;

    /// <summary>
    /// ODBC connection information.
    /// </summary>
    public OdbcConnection Connection
    {
        get { return connection; }
        set { connection = value; }
    }

    /// <summary>
    /// Constructor that takes one argument
    /// </summary>
    /// <param name="driveInfo">Drive provided by this provider</param>
    public AccessDBPSDriveInfo(PSDriveInfo driveInfo)
        : base(driveInfo)
    { }

} // class AccessDBPSDriveInfo

#endregion AccessDBPSDriveInfo

```

```
#region DatabaseTableInfo

    /// <summary>
    /// Contains information specific to the database table.
    /// Similar to the DirectoryInfo class.
    /// </summary>
    public class DatabaseTableInfo
    {
        /// <summary>
        /// Row from the "tables" schema
        /// </summary>
        public DataRow Data
        {
            get
            {
                return data;
            }
            set
            {
                data = value;
            }
        }
        private DataRow data;

        /// <summary>
        /// The table name.
        /// </summary>
        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }
        private String name;

        /// <summary>
        /// The number of rows in the table.
        /// </summary>
        public int RowCount
        {
            get
            {
                return rowCount;
            }
            set
            {
                rowCount = value;
            }
        }
    }
```

```

private int rowCount;

/// <summary>
/// The column definitions for the table.
/// </summary>
public DataColumnCollection Columns
{
    get
    {
        return columns;
    }
    set
    {
        columns = value;
    }
}
private DataColumnCollection columns;

/// <summary>
/// Constructor.
/// </summary>
/// <param name="row">The row definition.</param>
/// <param name="name">The table name.</param>
/// <param name="rowCount">The number of rows in the table.</param>
/// <param name="columns">Information on the column tables.</param>
public DatabaseTableInfo(DataRow row, string name, int rowCount,
                         DataColumnCollection columns)
{
    Name = name;
    Data = row;
    RowCount = rowCount;
    Columns = columns;
} // DatabaseTableInfo
} // class DatabaseTableInfo

#endregion DatabaseTableInfo

#region DatabaseRowInfo

/// <summary>
/// Contains information specific to an individual table row.
/// Analogous to the FileInfo class.
/// </summary>
public class DatabaseRowInfo
{
    /// <summary>
    /// Row data information.
    /// </summary>
    public DataRow Data
    {
        get
        {
            return data;
        }
        set
    }
}

```

```

        {
            data = value;
        }
    }

private DataRow data;

/// <summary>
/// The row index.
/// </summary>
public string RowNumber
{
    get
    {
        return rowNum;
    }
    set
    {
        rowNum = value;
    }
}
private string rowNum;

/// <summary>
/// Constructor.
/// </summary>
/// <param name="row">The row information.</param>
/// <param name="name">The row index.</param>
public DatabaseRowInfo(DataRow row, string name)
{
    RowNumber = name;
    Data = row;
} // DatabaseRowInfo
} // class DatabaseRowInfo

#endregion DatabaseRowInfo

#endregion Helper Classes
}

```

See Also

[System.Management.Automation.Provider.ItemCmdletProvider](#)

[System.Management.Automation.Provider.ContainerCmdletProvider](#)

[System.Management.Automation.Provider.NavigationCmdletProvider](#)

[Designing Your Windows PowerShell Provider](#)

AccessDBProviderSample05

Article • 03/24/2025

This sample shows how to overwrite container methods to support calls to the `Move-Item` and `Join-Path` cmdlets. These methods should be implemented when the user needs to move items within a container and if the data store contains nested containers. The provider class in this sample derives from the [System.Management.Automation.Provider.NavigationCmdletProvider](#) class.

Demonstrates

Important

Your provider class will most likely derive from one of the following classes and possibly implement other provider interfaces:

- [System.Management.Automation.Provider.ItemCmdletProvider](#) class. See [AccessDBProviderSample03](#).
- [System.Management.Automation.Provider.ContainerCmdletProvider](#) class. See [AccessDBProviderSample04](#).
- [System.Management.Automation.Provider.NavigationCmdletProvider](#) class.

For more information about choosing which provider class to derive from based on provider features, see [Designing Your Windows PowerShell Provider](#).

This sample demonstrates the following:

- Declaring the `CmdletProvider` attribute.
- Defining a provider class that derives from the [System.Management.Automation.Provider.NavigationCmdletProvider](#) class.
- Overwriting the [System.Management.Automation.Provider.NavigationCmdletProvider.MoveItem*](#) method to change the behavior of the `Move-Item` cmdlet, allowing the user to move items from one location to another. (This sample does not show how to add dynamic parameters to the `Move-Item` cmdlet.)
- Overwriting the [System.Management.Automation.Provider.NavigationCmdletProvider.MakePath*](#)

method to change the behavior of the `Join-Path` cmdlet.

- Overwriting the `System.Management.Automation.Provider.NavigationCmdletProvider.I.setItemContainer*` method.
- Overwriting the `System.Management.Automation.Provider.NavigationCmdletProvider.GetChildName*` method.
- Overwriting the `System.Management.Automation.Provider.NavigationCmdletProvider.GetParentPath*` method.
- Overwriting the `System.Management.Automation.Provider.NavigationCmdletProvider.NormalizeRelativePath*` method.

Example

This sample shows how to overwrite the methods needed to move items in a Microsoft Access data base.

C#

```
using System;
using System.IO;
using System.Data;
using System.Data.Odbc;
using System.Diagnostics;
using System.Collections.ObjectModel;
using System.Text;
using System.Text.RegularExpressions;
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.ComponentModel;
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

        /// <summary>
        /// This example implements the navigation methods.
        /// </summary>
        [CmdletProvider("AccessDB", ProviderCapabilities.None)]
        public class AccessDBProvider : NavigationCmdletProvider
    {
```

```

#region Drive Manipulation

    /// <summary>
    /// Create a new drive. Create a connection to the database file and
set
    /// the Connection property in the PSDriveInfo.
    /// </summary>
    /// <param name="drive">
    /// Information describing the drive to add.
    /// </param>
    /// <returns>The added drive.</returns>
protected override PSDriveInfo NewDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            null)
        );
        return null;
    }

    // check if drive root is not null or empty
    // and if its an existing file
    if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root)
== false))
    {
        WriteError(new ErrorRecord(
            new ArgumentException("drive.Root"),
            "NoRoot",
            ErrorCategory.InvalidArgument,
            drive)
        );
        return null;
    }

    // create a new drive and create an ODBC connection to the new
drive
    AccessDBPSDriveInfo accessDBPSDriveInfo = new
AccessDBPSDriveInfo(drive);

    OdbcConnectionStringBuilder builder = new
OdbcConnectionStringBuilder();

    builder.Driver = "Microsoft Access Driver (*.mdb)";
    builder.Add("DBQ", drive.Root);

    OdbcConnection conn = new

```

```

OdbcConnection(builder.ConnectionString);
    conn.Open();
    accessDBPSDriveInfo.Connection = conn;

        return accessDBPSDriveInfo;
} // NewDrive

/// <summary>
/// Removes a drive from the provider.
/// </summary>
/// <param name="drive">The drive to remove.</param>
/// <returns>The drive removed.</returns>
protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            drive)
        );
        return null;
    }

    // close ODBC connection to the drive
    AccessDBPSDriveInfo accessDBPSDriveInfo = drive as
AccessDBPSDriveInfo;

    if (accessDBPSDriveInfo == null)
    {
        return null;
    }
    accessDBPSDriveInfo.Connection.Close();

    return accessDBPSDriveInfo;
} // RemoveDrive

#endregion Drive Manipulation

#region Item Methods

/// <summary>
/// Retrieves an item using the specified path.
/// </summary>
/// <param name="path">The path to the item to return.</param>
protected override void GetItem(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        WriteItemObject(this.PSDriveInfo, path, true);
        return;
    }
}

```

```

} // if (PathIsDrive...

// Get table name and row information from the path and do
// necessary actions
string tableName;
int rowNumber;

PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

if (type == PathType.Table)
{
    DatabaseTableInfo table = GetTable(tableName);
    WriteItemObject(table, path, true);
}
else if (type == PathType.Row)
{
    DatabaseRowInfo row = GetRow(tableName, rowNumber);
    WriteItemObject(row, path, false);
}
else
{
    ThrowTerminatingInvalidOperationException(path);
}

} // GetItem

/// <summary>
/// Set the content of a row of data specified by the supplied path
/// parameter.
/// </summary>
/// <param name="path">Specifies the path to the row whose columns
/// will be updated.</param>
/// <param name="values">Comma separated string of values</param>
protected override void SetItem(string path, object values)
{
    // Get type, table name and row number from the path specified
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type != PathType.Row)
    {
        WriteError(new ErrorRecord(new NotSupportedException(
            "SetNotSupported"), "",
            ErrorCategory.InvalidOperation, path));

        return;
    }

    // Get in-memory representation of table
    OdbcDataAdapter da = GetAdapterForTable(tableName);
}

```

```

    if (da == null)
    {
        return;
    }
    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    if (rowNumber >= table.Rows.Count)
    {
        // The specified row number has to be available. If not
        // NewItem has to be used to add a new row
        throw new ArgumentException("Row specified is not
available");
    } // if (rowNum...

    string[] colValues = (values as string).Split(',');
}

// set the specified row
DataRow row = table.Rows[rowNumber];

for (int i = 0; i < colValues.Length; i++)
{
    row[i] = colValues[i];
}

// Update the table
if (ShouldProcess(path, "SetItem"))
{
    da.Update(ds, tableName);
}

} // SetItem

/// <summary>
/// Test to see if the specified item exists.
/// </summary>
/// <param name="path">The path to the item to verify.</param>
/// <returns>True if the item is found.</returns>
protected override bool ItemExists(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        return true;
    }

    // Obtain type, table name and row number from path
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    DatabaseTableInfo table = GetTable(tableName);
}

```

```

        if (type == PathType.Table)
        {
            // if specified path represents a table then
DatabaseTableInfo
            // object for the same should exist
            if (table != null)
            {
                return true;
            }
        }
        else if (type == PathType.Row)
        {
            // if specified path represents a row then DatabaseTableInfo
should
            // exist for the table and then specified row number must be
within
            // the maximum row count in the table
            if (table != null && rowNum < table.RowCount)
            {
                return true;
            }
        }

        return false;
    } // ItemExists

    /// <summary>
    /// Test to see if the specified path is syntactically valid.
    /// </summary>
    /// <param name="path">The path to validate.</param>
    /// <returns>True if the specified path is valid.</returns>
protected override bool IsValidPath(string path)
{
    bool result = true;

    // check if the path is null or empty
    if (String.IsNullOrEmpty(path))
    {
        result = false;
    }

    // convert all separators in the path to a uniform one
    path = NormalizePath(path);

    // split the path into individual chunks
    string[] pathChunks = path.Split(pathSeparator.ToCharArray());

    foreach (string pathChunk in pathChunks)
    {
        if (pathChunk.Length == 0)
        {
            result = false;
        }
    }
}

```

```

        return result;
    } // IsValidPath

    #endregion Item Overloads

    #region Container Overloads

    /// <summary>
    /// Return either the tables in the database or the datarows
    /// </summary>
    /// <param name="path">The path to the parent</param>
    /// <param name="recurse">True to return all child items recursively.
    /// </param>
    protected override void GetChildItems(string path, bool recurse)
    {
        // If path represented is a drive then the children in the path
        // are
        // tables. Hence all tables in the drive represented will have to
        // be
        // returned
        if (PathIsDrive(path))
        {
            foreach (DatabaseTableInfo table in GetTables())
            {
                WriteItemObject(table, path, true);

                // if the specified item exists and recurse has been set
                // then
                // all child items within it have to be obtained as well
                if (ItemExists(path) && recurse)
                {
                    GetChildItems(path + pathSeparator + table.Name,
                    recurse);
                }
            } // foreach (DatabaseTableInfo...
        } // if (PathIsDrive...
        else
        {
            // Get the table name, row number and type of path from the
            // path specified
            string tableName;
            int rowNumber;

            PathType type = GetNamesFromPath(path, out tableName, out
            rowNumber);

            if (type == PathType.Table)
            {
                // Obtain all the rows within the table
                foreach (DatabaseRowInfo row in GetRows(tableName))
                {
                    WriteItemObject(row, path + pathSeparator +
row.RowNumber,
                        false);
                } // foreach (DatabaseRowInfo...
            }
        }
    }
}

```

```

        }
        else if (type == PathType.Row)
        {
            // In this case the user has directly specified a row,
            hence
            // just give that particular row
            DatabaseRowInfo row = GetRow(tableName, rowNumber);
            WriteItemObject(row, path + pathSeparator +
            row.RowNumber,
                            false);
        }
        else
        {
            // In this case, the path specified is not valid
            ThrowTerminatingInvalidPathException(path);
        }
    } // else
} // GetChildItems

/// <summary>
/// Return the names of all child items.
/// </summary>
/// <param name="path">The root path.</param>
/// <param name="returnContainers">Not used.</param>
protected override void GetChildNames(string path,
                                         ReturnContainers returnContainers)
{
    // If the path represented is a drive, then the child items are
    // tables. get the names of all the tables in the drive.
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table.Name, path, true);
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get type, table name and row number from path specified
        string tableName;
        int rowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Table)
        {
            // Get all the rows in the table and then write out the
            // row numbers.
            foreach (DatabaseRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row.RowNumber, path, false);
            } // foreach (DatabaseRowInfo...
        }
        else if (type == PathType.Row)
    }
}

```

```

        {
            // In this case the user has directly specified a row,
hence
            // just give that particular row
            DatabaseRowInfo row = GetRow(tableName, rowNumber);

            WriteItemObject(row.RowNumber, path, false);
        }
        else
        {
            ThrowTerminatingInvalidOperationException(path);
        }
    } // else
} // GetChildNames

/// <summary>
/// Determines if the specified path has child items.
/// </summary>
/// <param name="path">The path to examine.</param>
/// <returns>
/// True if the specified path has child items.
/// </returns>
protected override bool HasChildItems(string path)
{
    if (PathIsDrive(path))
    {
        return true;
    }

    return (ChunkPath(path).Length == 1);
} // HasChildItems

/// <summary>
/// Creates a new item at the specified path.
/// </summary>
///
/// <param name="path">
/// The path to the new item.
/// </param>
///
/// <param name="type">
/// Type for the object to create. "Table" for creating a new table
and
/// "Row" for creating a new row in a table.
/// </param>
///
/// <param name="newValue">
/// Object for creating new instance of a type at the specified path.
For
/// creating a "Table" the object parameter is ignored and for
creating
/// a "Row" the object must be of type string which will contain
comma
/// separated values of the rows to insert.
/// </param>

```

```

protected override void NewItem(string path, string type,
                               object newItemValue)
{
    string tableName;
    int rowNumber;

    PathType pt = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (pt == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }

    // Check if type is either "table" or "row", if not throw an
    // exception
    if (!String.Equals(type, "table",
 StringComparison.OrdinalIgnoreCase)
        && !String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
    {
        WriteError(new ErrorRecord
                    (new ArgumentException("Type must be either
a table or row"),
                     "CannotCreateSpecifiedObject",
                     ErrorCategory.InvalidArgument,
                     path
                    )
                );
    }

    throw new ArgumentException("This provider can only create
items of type \"table\" or \"row\"");
}

// Path type is the type of path of the container. So if a drive
// is specified, then a table can be created under it and if a
table
of
by
row
based
// is specified, then a row can be created under it. For the sake
// completeness, if a row is specified, then if the row specified
// the path does not exist, a new row is created. However, the
// number may not match as the row numbers only get incremented
// on the number of rows

if (PathIsDrive(path))
{
    if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
    {
        // Execute command using ODBC connection to create a
table
        try

```

```

    {
        // create the table using an sql statement
        string newTableName = newItemValue.ToString();
        string sql = "create table " + newTableName
                    + " (ID INT)";

        // Create the table using the Odbc connection from
        the
        // drive.
        AccessDBPSDriveInfo di = this.PSDriveInfo as
        AccessDBPSDriveInfo;

        if (di == null)
        {
            return;
        }
        OdbcConnection connection = di.Connection;

        if (ShouldProcess(newTableName, "create"))
        {
            OdbcCommand cmd = new OdbcCommand(sql,
connection);
            cmd.ExecuteScalar();
        }
        catch (Exception ex)
        {
            WriteError(new ErrorRecord(ex,
"CannotCreateSpecifiedTable",
ErrorCategory.InvalidOperation, path)
);
        }
    } // if (String...
    else if (String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
    {
        throw new
            ArgumentException("A row cannot be created under a
database, specify a path that represents a Table");
    }
} // if (PathIsDrive...
else
{
    if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
    {
        if (rowNumber < 0)
        {
            throw new
                ArgumentException("A table cannot be created
within another table, specify a path that represents a database");
        }
        else
        {
            throw new

```

```

        ArgumentException("A table cannot be created
inside a row, specify a path that represents a database");
    }
} //if (String.Equals....
// if path specified is a row, create a new row
else if (String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
{
    // The user is required to specify the values to be
inserted
    // into the table in a single string separated by commas
    string value = newItemValue as string;

    if (String.IsNullOrEmpty(value))
    {
        throw new
            ArgumentException("Value argument must have comma
separated values of each column in a row");
    }
    string[] rowValues = value.Split(',');
}

OdbcDataAdapter da = GetAdapterForTable(tableName);

if (da == null)
{
    return;
}

DataSet ds = GetDataSetForTable(da, tableName);
DataTable table = GetDataTable(ds, tableName);

if (rowValues.Length != table.Columns.Count)
{
    string message =
String.Format(CultureInfo.CurrentCulture,
                "The table has {0} columns and
the value specified must have so many comma separated values",
                table.Columns.Count);

    throw new ArgumentException(message);
}

if (!Force && (rowNumber >= 0 && rowNumber <
table.Rows.Count))
{
    string message =
String.Format(CultureInfo.CurrentCulture,
                "The row {0} already exists. To
create a new row specify row number as {1}, or specify path to a table, or
use the -Force parameter",
                rowNumber,
                table.Rows.Count);

    throw new ArgumentException(message);
}

```

```

        if (rowNumber > table.Rows.Count)
        {
            string message =
String.Format(CultureInfo.CurrentCulture,
                           "To create a new row specify row
number as {0}, or specify path to a table",
                           table.Rows.Count);

            throw new ArgumentException(message);
        }

        // Create a new row and update the row with the input
        // provided by the user
        DataRow row = table.NewRow();
        for (int i = 0; i < rowValues.Length; i++)
        {
            row[i] = rowValues[i];
        }
        table.Rows.Add(row);

        if (ShouldProcess(tableName, "update rows"))
        {
            // Update the table from memory back to the data
source
            da.Update(ds, tableName);
        }

        } // else if (String...
    } // else ...

} // NewItem

/// <summary>
/// Copies an item at the specified path to the location specified
/// </summary>
///
/// <param name="path">
/// Path of the item to copy
/// </param>
///
/// <param name="copyPath">
/// Path of the item to copy to
/// </param>
///
/// <param name="recurse">
/// Tells the provider to recurse subcontainers when copying
/// </param>
///
protected override void CopyItem(string path, string copyPath, bool
recurse)
{
    string tableName, copyTableName;
    int pageNumber, copyRowNumber;

```

```

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);
        PathType copyType = GetNamesFromPath(copyPath, out copyTableName,
out copyRowNumber);

        if (type == PathType.Invalid)
        {
            ThrowTerminatingInvalidPathException(path);
        }

        if (type == PathType.Invalid)
        {
            ThrowTerminatingInvalidPathException(copyPath);
        }

        // Get the table and the table to copy to
        OdbcDataAdapter da = GetAdapterForTable(tableName);
        if (da == null)
        {
            return;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        OdbcDataAdapter cda = GetAdapterForTable(copyTableName);
        if (cda == null)
        {
            return;
        }

        DataSet cds = GetDataSetForTable(cda, copyTableName);
        DataTable copyTable = GetDataTable(cds, copyTableName);

        // if source represents a table
        if (type == PathType.Table)
        {
            // if copyPath does not represent a table
            if (copyType != PathType.Table)
            {
                ArgumentException e = new ArgumentException("Table can
only be copied on to another table location");

                WriteError(new ErrorRecord(e, "PathNotValid",
                    ErrorCategory.InvalidArgument, copyPath));

                throw e;
            }

            // if table already exists then force parameter should be set
            // to force a copy
            if (!Force && GetTable(copyTableName) != null)
            {
                throw new ArgumentException("Specified path already
exists");
            }
        }
    }
}

```

```

        }

        for (int i = 0; i < table.Rows.Count; i++)
        {
            DataRow row = table.Rows[i];
            DataRow copyRow = copyTable.NewRow();

            copyRow.ItemArray = row.ItemArray;
            copyTable.Rows.Add(copyRow);
        }
    } // if (type == ...
// if source represents a row
else
{
    if (copyType == PathType.Row)
    {
        if (!Force && (copyRowNumber < copyTable.Rows.Count))
        {
            throw new ArgumentException("Specified path already
exists.");
        }

        DataRow row = table.Rows[rowNumber];
        DataRow copyRow = null;

        if (copyRowNumber < copyTable.Rows.Count)
        {
            // copy to an existing row
            copyRow = copyTable.Rows[copyRowNumber];
            copyRow.ItemArray = row.ItemArray;
            copyRow[0] = GetNextID(copyTable);
        }
        else if (copyRowNumber == copyTable.Rows.Count)
        {
            // copy to the next row in the table that will
            // be created
            copyRow = copyTable.NewRow();
            copyRow.ItemArray = row.ItemArray;
            copyRow[0] = GetNextID(copyTable);
            copyTable.Rows.Add(copyRow);
        }
        else
        {
            // attempting to copy to a nonexistent row or a row
            // that cannot be created now - throw an exception
            string message =
String.Format(CultureInfo.CurrentCulture,
                         "The item cannot be specified to
the copied row. Specify row number as {0}, or specify a path to the table.",
                         table.Rows.Count);

            throw new ArgumentException(message);
        }
    }
    else

```

```

        {
            // destination path specified represents a table,
            // create a new row and copy the item
            DataRow copyRow = copyTable.NewRow();
            copyRow.ItemArray = table.Rows[rowNumber].ItemArray;
            copyRow[0] = GetNextID(copyTable);
            copyTable.Rows.Add(copyRow);
        }
    }

    if (ShouldProcess(copyTableName, "CopyItems"))
    {
        cda.Update(cds, copyTableName);
    }

} //CopyItem

/// <summary>
/// Removes (deletes) the item at the specified path
/// </summary>
///
/// <param name="path">
/// The path to the item to remove.
/// </param>
///
/// <param name="recurse">
/// True if all children in a subtree should be removed, false if
only
    /// the item at the specified path should be removed. Is applicable
    /// only for container (table) items. Its ignored otherwise (even if
    /// specified).
    /// </param>
    ///
    /// <remarks>
    /// There are no elements in this store which are hidden from the
user.
    /// Hence this method will not check for the presence of the Force
    /// parameter
    /// </remarks>
    ///
protected override void RemoveItem(string path, bool recurse)
{
    string tableName;
    int rowNumber = 0;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        // if recurse flag has been specified, delete all the rows as
well
        if (recurse)
        {
            OdbcDataAdapter da = GetAdapterForTable(tableName);

```

```

    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    for (int i = 0; i < table.Rows.Count; i++)
    {
        table.Rows[i].Delete();
    }

    if (ShouldProcess(path, "RemoveItem"))
    {
        da.Update(ds, tableName);
        RemoveTable(tableName);
    }
    } //if (recurse...
    else
    {
        // Remove the table
        if (ShouldProcess(path, "RemoveItem"))
        {
            RemoveTable(tableName);
        }
    }
}
else if (type == PathType.Row)
{
    OdbcDataAdapter da = GetAdapterForTable(tableName);
    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    table.Rows[rowNumber].Delete();

    if (ShouldProcess(path, "RemoveItem"))
    {
        da.Update(ds, tableName);
    }
}
else
{
    ThrowTerminatingInvalidOperationException(path);
}

} // RemoveItem

#endregion Container Overloads

```

```

#region Navigation

/// <summary>
/// Determine if the path specified is that of a container.
/// </summary>
/// <param name="path">The path to check.</param>
/// <returns>True if the path specifies a container.</returns>
protected override bool IsItemContainer(string path)
{
    if (PathIsDrive(path))
    {
        return true;
    }

    string[] pathChunks = ChunkPath(path);
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        foreach (DatabaseTableInfo ti in GetTables())
        {
            if (string.Equals(ti.Name, tableName,
 StringComparison.OrdinalIgnoreCase))
            {
                return true;
            }
        } // foreach (DatabaseTableInfo...
    } // if (pathChunks...

    return false;
} // IsItemContainer

/// <summary>
/// Get the name of the leaf element in the specified path
/// </summary>
///
/// <param name="path">
/// The full or partial provider specific path
/// </param>
///
/// <returns>
/// The leaf element in the path
/// </returns>
protected override string GetChildName(string path)
{
    if (PathIsDrive(path))
    {
        return path;
    }

    string tableName;

```

```
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        return tableName;
    }
    else if (type == PathType.Row)
    {
        return rowNumber.ToString(CultureInfo.CurrentCulture);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }

    return null;
}

/// <summary>
/// Removes the child segment of the path and returns the remaining
/// parent portion
/// </summary>
///
/// <param name="path">
/// A full or partial provider specific path. The path may be to an
/// item that may or may not exist.
/// </param>
///
/// <param name="root">
/// The fully qualified path to the root of a drive. This parameter
/// may be null or empty if a mounted drive is not in use for this
/// operation. If this parameter is not null or empty the result
/// of the method should not be a path to a container that is a
/// parent or in a different tree than the root.
/// </param>
///
/// <returns></returns>

protected override string GetParentPath(string path, string root)
{
    // If root is specified then the path has to contain
    // the root. If not nothing should be returned
    if (!String.IsNullOrEmpty(root))
    {
        if (!path.Contains(root))
        {
            return null;
        }
    }

    return path.Substring(0, path.LastIndexOf(pathSeparator,
 StringComparison.OrdinalIgnoreCase));
}
```

```
}

/// <summary>
/// Joins two strings with a provider specific path separator.
/// </summary>
///
/// <param name="parent">
/// The parent segment of a path to be joined with the child.
/// </param>
///
/// <param name="child">
/// The child segment of a path to be joined with the parent.
/// </param>
///
/// <returns>
/// A string that represents the parent and child segments of the
path
/// joined by a path separator.
/// </returns>

protected override string MakePath(string parent, string child)
{
    string result;

    string normalParent = NormalizePath(parent);
    normalParent = RemoveDriveFromPath(normalParent);
    string normalChild = NormalizePath(child);
    normalChild = RemoveDriveFromPath(normalChild);

    if (String.IsNullOrEmpty(normalParent) &&
String.IsNullOrEmpty(normalChild))
    {
        result = String.Empty;
    }
    else if (String.IsNullOrEmpty(normalParent) &&
!String.IsNullOrEmpty(normalChild))
    {
        result = normalChild;
    }
    else if (!String.IsNullOrEmpty(normalParent) &&
String.IsNullOrEmpty(normalChild))
    {
        if (normalParent.EndsWith(pathSeparator,
 StringComparison.OrdinalIgnoreCase))
        {
            result = normalParent;
        }
        else
        {
            result = normalParent + pathSeparator;
        }
    } // else if (!String...
    else
    {
        if (!normalParent.Equals(String.Empty) &&
```

```

                !normalParent.EndsWith(pathSeparator,
StringComparison.OrdinalIgnoreCase))
{
    result = normalParent + pathSeparator;
}
else
{
    result = normalParent;
}

    if (normalChild.StartsWith(pathSeparator,
StringComparison.OrdinalIgnoreCase))
{
    result += normalChild.Substring(1);
}
else
{
    result += normalChild;
}
} // else

return result;
} // MakePath

/// <summary>
/// Normalizes the path that was passed in and returns the normalized
/// path as a relative path to the basePath that was passed.
/// </summary>
///
/// <param name="path">
/// A fully qualified provider specific path to an item. The item
/// should exist or the provider should write out an error.
/// </param>
///
/// <param name="basepath">
/// The path that the return value should be relative to.
/// </param>
///
/// <returns>
/// A normalized path that is relative to the basePath that was
/// passed. The provider should parse the path parameter, normalize
/// the path, and then return the normalized path relative to the
/// basePath.
/// </returns>

protected override string NormalizeRelativePath(string path,
                                                string basepath)
{
    // Normalize the paths first
    string normalPath = NormalizePath(path);
    normalPath = RemoveDriveFromPath(normalPath);
    string normalBasePath = NormalizePath(basepath);
    normalBasePath = RemoveDriveFromPath(normalBasePath);

    if (String.IsNullOrEmpty(normalBasePath))

```

```

        {
            return normalPath;
        }
        else
        {
            if (!normalPath.Contains(normalbasePath))
            {
                return null;
            }

            return normalPath.Substring(normalbasePath.Length +
pathSeparator.Length);
        }
    }

    /// <summary>
    /// Moves the item specified by the path to the specified destination
    /// </summary>
    ///
    /// <param name="path">
    /// The path to the item to be moved
    /// </param>
    ///
    /// <param name="destination">
    /// The path of the destination container
    /// </param>

    protected override void MoveItem(string path, string destination)
{
    // Get type, table name and rowNumber from the path
    string tableName, destTableName;
    int rowNumber, destRowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    PathType destType = GetNamesFromPath(destination, out
destTableName,
                                         out destRowNumber);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }

    if (destType == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(destination);
    }

    if (type == PathType.Table)
    {
        ArgumentException e = new ArgumentException("Move not
supported for tables");

```

```

        WriteError(new ErrorRecord(e, "MoveNotSupported",
            ErrorCategory.InvalidArgument, path));

        throw e;
    }
    else
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);
        if (da == null)
        {
            return;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        OdbcDataAdapter dda = GetAdapterForTable(destTableName);
        if (dda == null)
        {
            return;
        }

        DataSet dds = GetDataSetForTable(dda, destTableName);
        DataTable destTable = GetDataTable(dds, destTableName);
        DataRow row = table.Rows[rowNumber];

        if (destType == PathType.Table)
        {
            DataRow destRow = destTable.NewRow();

            destRow.ItemArray = row.ItemArray;
        }
        else
        {
            DataRow destRow = destTable.Rows[destRowNumber];

            destRow.ItemArray = row.ItemArray;
        }

        // Update the changes
        if (ShouldProcess(path, "MoveItem"))
        {
            WriteItemObject(row, path, false);
            dda.Update(dds, destTableName);
        }
    }
}

#endregion Navigation

#region Helper Methods

/// <summary>
/// Checks if a given path is actually a drive name.
/// </summary>

```

```

    ///<param name="path">The path to check.</param>
    ///<returns>
    /// True if the path given represents a drive, false otherwise.
    ///</returns>
    private bool PathIsDrive(string path)
    {
        // Remove the drive name and first path separator. If the
        // path is reduced to nothing, it is a drive. Also if its
        // just a drive then there wont be any path separators
        if (String.IsNullOrEmpty(
            path.Replace(this.PSDriveInfo.Root, "")) ||
            String.IsNullOrEmpty(
                path.Replace(this.PSDriveInfo.Root + pathSeparator,
                "")))
        {
            return true;
        }
        else
        {
            return false;
        }
    } // PathIsDrive

    ///<summary>
    /// Breaks up the path into individual elements.
    ///</summary>
    ///<param name="path">The path to split.</param>
    ///<returns>An array of path segments.</returns>
    private string[] ChunkPath(string path)
    {
        // Normalize the path before splitting
        string normalPath = NormalizePath(path);

        // Return the path with the drive name and first path
        // separator character removed, split by the path separator.
        string pathNoDrive = normalPath.Replace(this.PSDriveInfo.Root
            + pathSeparator, "");

        return pathNoDrive.Split(pathSeparator.ToCharArray());
    } // ChunkPath

    ///<summary>
    /// Adapts the path, making sure the correct path separator
    /// character is used.
    ///</summary>
    ///<param name="path"></param>
    ///<returns></returns>
    private string NormalizePath(string path)
    {
        string result = path;

        if (!String.IsNullOrEmpty(path))
        {

```

```

        result = path.Replace("/", pathSeparator);
    }

    return result;
} // NormalizePath

/// <summary>
/// Ensures that the drive is removed from the specified path
/// </summary>
///
/// <param name="path">Path from which drive needs to be
removed</param>
/// <returns>Path with drive information removed</returns>
private string RemoveDriveFromPath(string path)
{
    string result = path;
    string root;

    if (this.PSDriveInfo == null)
    {
        root = String.Empty;
    }
    else
    {
        root = this.PSDriveInfo.Root;
    }

    if (result == null)
    {
        result = String.Empty;
    }

    if (result.Contains(root))
    {
        result = result.Substring(result.IndexOf(root,
StringComparison.OrdinalIgnoreCase) + root.Length);
    }

    return result;
}

/// <summary>
/// Chunks the path and returns the table name and the row number
/// from the path
/// </summary>
/// <param name="path">Path to chunk and obtain information</param>
/// <param name="tableName">Name of the table as represented in the
/// path</param>
/// <param name="rowNumber">Row number obtained from the path</param>
/// <returns>what the path represents</returns>
private PathType GetNamesFromPath(string path, out string tableName,
out int rowNumber)
{
    PathType retVal = PathType.Invalid;
    rowNumber = -1;
}

```

```

tableName = null;

// Check if the path specified is a drive
if (PathIsDrive(path))
{
    return PathType.Database;
}

// chunk the path into parts
string[] pathChunks = ChunkPath(path);

switch (pathChunks.Length)
{
    case 1:
    {
        string name = pathChunks[0];

        if (TableNameIsValid(name))
        {
            tableName = name;
            retVal = PathType.Table;
        }
    }
    break;

    case 2:
    {
        string name = pathChunks[0];

        if (TableNameIsValid(name))
        {
            tableName = name;
        }

        int number = SafeConvertRowNumber(pathChunks[1]);

        if (number >= 0)
        {
            rowCount = number;
            retVal = PathType.Row;
        }
        else
        {
            WriteError(new ErrorRecord(
                new ArgumentException("Row number is not
valid"),
                "RowNumberNotValid",
                ErrorCategory.InvalidArgument,
                path));
        }
    }
    break;

    default:
    {

```

```

        WriteError(new ErrorRecord(
            new ArgumentException("The path supplied has too
many segments"),
            "PathNotValid",
            ErrorCategory.InvalidArgument,
            path));
    }
    break;
} // switch(pathChunks...

return retVal;
} // GetNamesFromPath

/// <summary>
/// Throws an argument exception stating that the specified path does
/// not represent either a table or a row
/// </summary>
/// <param name="path">path which is invalid</param>
private void ThrowTerminatingInvalidPathException(string path)
{
    StringBuilder message = new StringBuilder("Path must represent
either a table or a row :");
    message.Append(path);

    throw new ArgumentException(message.ToString());
}

/// <summary>
/// Retrieve the list of tables from the database.
/// </summary>
/// <returns>
/// Collection of DatabaseTableInfo objects, each object representing
/// information about one database table
/// </returns>
private Collection<DatabaseTableInfo> GetTables()
{
    Collection<DatabaseTableInfo> results =
        new Collection<DatabaseTableInfo>();

    // using ODBC connection to the database and get the schema of
tables
    AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

    if (di == null)
    {
        return null;
    }

    OdbcConnection connection = di.Connection;
    DataTable dt = connection.GetSchema("Tables");
    int count;

    // iterate through all rows in the schema and create
DatabaseTableInfo
    // objects which represents a table
}

```

```

        foreach (DataRow dr in dt.Rows)
    {
        String tableName = dr["TABLE_NAME"] as String;
        DataColumnCollection columns = null;

        // find the number of rows in the table
        try
        {
            String cmd = "Select count(*) from \\" + tableName + "\\";
            OdbcCommand command = new OdbcCommand(cmd, connection);

            count = (Int32)command.ExecuteScalar();
        }
        catch
        {
            count = 0;
        }

        // create DatabaseTableInfo object representing the table
        DatabaseTableInfo table =
            new DatabaseTableInfo(dr, tableName, count, columns);

        results.Add(table);
    } // foreach (DataRow...

        return results;
} // GetTables

/// <summary>
/// Return row information from a specified table.
/// </summary>
/// <param name="tableName">The name of the database table from
/// which to retrieve rows.</param>
/// <returns>Collection of row information objects.</returns>
private Collection<DatabaseRowInfo> GetRows(string tableName)
{
    Collection<DatabaseRowInfo> results =
        new Collection<DatabaseRowInfo>();

    // Obtain rows in the table and add it to the collection
    try
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);

        if (da == null)
        {
            return null;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        int i = 0;
        foreach (DataRow row in table.Rows)
        {

```

```

        results.Add(new DatabaseRowInfo(row,
i.ToString(CultureInfo.CurrentCulture)));
        i++;
    } // foreach (DataRow...
}
catch (Exception e)
{
    WriteError(new ErrorRecord(e, "CannotAccessSpecifiedRows",
ErrorCategory.InvalidOperation, tableName));
}

return results;
} // GetRows

/// <summary>
/// Retrieve information about a single table.
/// </summary>
/// <param name="tableName">The table for which to retrieve
/// data.</param>
/// <returns>Table information.</returns>
private DatabaseTableInfo GetTable(string tableName)
{
    foreach (DatabaseTableInfo table in GetTables())
    {
        if (String.Equals(tableName, table.Name,
 StringComparison.OrdinalIgnoreCase))
        {
            return table;
        }
    }

    return null;
} // GetTable

/// <summary>
/// Removes the specified table from the database
/// </summary>
/// <param name="tableName">Name of the table to remove</param>
private void RemoveTable(string tableName)
{
    // validate if tablename is valid and if table is present
    if (String.IsNullOrEmpty(tableName) ||
!TableNameIsValid(tableName) || !TableIsPresent(tableName))
    {
        return;
    }

    // Execute command using ODBC connection to remove a table
    try
    {
        // delete the table using an sql statement
        string sql = "drop table " + tableName;

        AccessDBPSDriveInfo di = this.PSDriveInfo as

```

```

AccessDBPSDriveInfo;

    if (di == null)
    {
        return;
    }
    OdbcConnection connection = di.Connection;

    OdbcCommand cmd = new OdbcCommand(sql, connection);
    cmd.ExecuteScalar();
}
catch (Exception ex)
{
    WriteError(new ErrorRecord(ex, "CannotRemoveSpecifiedTable",
        ErrorCategory.InvalidOperation, null)
    );
}

} // RemoveTable

/// <summary>
/// Obtain a data adapter for the specified Table
/// </summary>
/// <param name="tableName">Name of the table to obtain the
/// adapter for</param>
/// <returns>Adapter object for the specified table</returns>
/// <remarks>An adapter serves as a bridge between a DataSet (in
memory
/// representation of table) and the data source</remarks>
private OdbcDataAdapter GetAdapterForTable(string tableName)
{
    OdbcDataAdapter da = null;
    AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

    if (di == null || !TableNameIsValid(tableName) ||
!TableIsPresent(tableName))
    {
        return null;
    }

    OdbcConnection connection = di.Connection;

    try
    {
        // Create a odbc data adpater. This can be sued to update the
        // data source with the records that will be created here
        // using data sets
        string sql = "Select * from " + tableName;
        da = new OdbcDataAdapter(new OdbcCommand(sql, connection));

        // Create a odbc command builder object. This will create sql
        // commands automatically for a single table, thus
        // eliminating the need to create new sql statements for
        // every operation to be done.
        OdbcCommandBuilder cmd = new OdbcCommandBuilder(da);
    }
}

```

```

// Set the delete cmd for the table here
sql = "Delete from " + tableName + " where ID = ?";
da.DeleteCommand = new OdbcCommand(sql, connection);

// Specify a DeleteCommand parameter based on the "ID"
// column
da.DeleteCommand.Parameters.Add(new OdbcParameter());
da.DeleteCommand.Parameters[0].SourceColumn = "ID";

// Create an InsertCommand based on the sql string
// Insert into "tablename" values (?,?,?) where
// ? represents a column in the table. Note that
// the number of ? will be equal to the number of
// columns
DataSet ds = new DataSet();

da.FillSchema(ds, SchemaType.Source);
ds.Locale = CultureInfo.InvariantCulture;

sql = "Insert into " + tableName + " values ( ";
for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
{
    sql += "?, ";
}
sql = sql.Substring(0, sql.Length - 2);
sql += ")";
da.InsertCommand = new OdbcCommand(sql, connection);

// Create parameters for the InsertCommand based on the
// captions of each column
for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
{
    da.InsertCommand.Parameters.Add(new OdbcParameter());
    da.InsertCommand.Parameters[i].SourceColumn =
        ds.Tables["Table"].Columns[i].Caption;
}

// Open the connection if its not already open
if (connection.State != ConnectionState.Open)
{
    connection.Open();
}
catch (Exception e)
{
    WriteError(new ErrorRecord(e, "CannotAccessSpecifiedTable",
        ErrorCategory.InvalidOperation, tableName));
}

return da;
} // GetAdapterForTable

/// <summary>

```

```

/// Gets the DataSet (in memory representation) for the table
/// for the specified adapter
/// </summary>
/// <param name="adapter">Adapter to be used for obtaining
/// the table</param>
/// <param name="tableName">Name of the table for which a
/// DataSet is required</param>
/// <returns>The DataSet with the filled in schema</returns>
private DataSet GetDataSetForTable( OdbcDataAdapter adapter, string
tableName)
{
    Debug.Assert(adapter != null);

    // Create a dataset object which will provide an in-memory
    // representation of the data being worked upon in the
    // data source.
    DataSet ds = new DataSet();

    // Create a table named "Table" which will contain the same
    // schema as in the data source.
    //adapter.FillSchema(ds, SchemaType.Source);
    adapter.Fill(ds, tableName);
    ds.Locale = CultureInfo.InvariantCulture;

    return ds;
} //GetDataSetForTable

/// <summary>
/// Get the DataTable object which can be used to operate on
/// for the specified table in the data source
/// </summary>
/// <param name="ds">DataSet object which contains the tables
/// schema</param>
/// <param name="tableName">Name of the table</param>
/// <returns>Corresponding DataTable object representing
/// the table</returns>
///
private DataTable GetDataTable(DataSet ds, string tableName)
{
    Debug.Assert(ds != null);
    Debug.Assert(tableName != null);

    DataTable table = ds.Tables[tableName];
    table.Locale = CultureInfo.InvariantCulture;

    return table;
} // GetDataTable

/// <summary>
/// Retrieves a single row from the named table.
/// </summary>
/// <param name="tableName">The table that contains the
/// numbered row.</param>
/// <param name="row">The index of the row to return.</param>
/// <returns>The specified table row.</returns>

```

```

    private DatabaseRowInfo GetRow(string tableName, int row)
    {
        Collection<DatabaseRowInfo> di = GetRows(tableName);

        // if the row is invalid write an appropriate error else return
the
        // corresponding row information
        if (row < di.Count && row >= 0)
        {
            return di[row];
        }
        else
        {
            WriteError(new ErrorRecord(
                new ItemNotFoundException(),
                "RowNotFound",
                ErrorCategory.ObjectNotFound,
                row.ToString(CultureInfo.CurrentCulture))
            );
        }

        return null;
    } // GetRow

    /// <summary>
    /// Method to safely convert a string representation of a row number
    /// into its Int32 equivalent
    /// </summary>
    /// <param name="rowNumberAsStr">String representation of the row
    /// number</param>
    /// <remarks>If there is an exception, -1 is returned</remarks>
    private int SafeConvertRowNumber(string rowNumberAsStr)
    {
        int rowNumber = -1;
        try
        {
            rowNumber = Convert.ToInt32(rowNumberAsStr,
CultureInfo.CurrentCulture);
        }
        catch (FormatException fe)
        {
            WriteError(new ErrorRecord(fe, "RowStringFormatNotValid",
                ErrorCategory.InvalidData, rowNumberAsStr));
        }
        catch (OverflowException oe)
        {
            WriteError(new ErrorRecord(oe,
"RowStringConversionToNumberFailed",
                ErrorCategory.InvalidData, rowNumberAsStr));
        }

        return rowNumber;
    } // SafeConvertRowNumber

    /// <summary>

```

```

    ///> <summary>
    ///> </summary>
    ///> <param name="tableName">Table name to validate</param>
    ///> <remarks>Helps to check for SQL injection attacks</remarks>
    private bool TableNameIsValid(string tableName)
    {
        Regex exp = new Regex(pattern, RegexOptions.Compiled |
        RegexOptions.IgnoreCase);

        if (exp.IsMatch(tableName))
        {
            return true;
        }
        WriteError(new ErrorRecord(
            new ArgumentException("Table name not valid"),
            "TableNameNotValid",
            ErrorCategory.InvalidArgument, tableName));
        return false;
    } // TableNameIsValid

    ///> <summary>
    ///> Checks to see if the specified table is present in the
    ///> database
    ///> </summary>
    ///> <param name="tableName">Name of the table to check</param>
    ///> <returns>true, if table is present, false otherwise</returns>
    private bool TableIsPresent(string tableName)
    {
        // using ODBC connection to the database and get the schema of
        tables
        AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;
        if (di == null)
        {
            return false;
        }

        OdbcConnection connection = di.Connection;
        DataTable dt = connection.GetSchema("Tables");

        // check if the specified tableName is available
        // in the list of tables present in the database
        foreach (DataRow dr in dt.Rows)
        {
            string name = dr["TABLE_NAME"] as string;
            if (name.Equals(tableName,
                StringComparison.OrdinalIgnoreCase))
            {
                return true;
            }
        }

        WriteError(new ErrorRecord(
            new ArgumentException("Specified Table is not present in
database"), "TableNotAvailable",
            ErrorCategory.InvalidArgument, tableName));
    }
}

```

```

        return false;
    } // TableIsPresent

    /// <summary>
    /// Gets the next available ID in the table
    /// </summary>
    /// <param name="table">DataTable object representing the table to
    /// search for ID</param>
    /// <returns>next available id</returns>
    private int GetNextID(DataTable table)
    {
        int big = 0;

        for (int i = 0; i < table.Rows.Count; i++)
        {
            DataRow row = table.Rows[i];

            int id = (int)row["ID"];

            if (big < id)
            {
                big = id;
            }
        }

        big++;
        return big;
    }

#endregion Helper Methods

#region Private Properties

private string pathSeparator = "\\";
private static string pattern = @"^+[a-z]+[0-9]*_*$";

private enum PathType { Database, Table, Row, Invalid };

#endregion Private Properties

} // AccessDBProvider

#endregion AccessDBProvider

#region Helper Classes

#region AccessDBPSDriveInfo

/// <summary>
/// Any state associated with the drive should be held here.
/// In this case, it's the connection to the database.
/// </summary>
internal class AccessDBPSDriveInfo : PSDriveInfo
{

```

```

private OdbcConnection connection;

/// <summary>
/// ODBC connection information.
/// </summary>
public OdbcConnection Connection
{
    get { return connection; }
    set { connection = value; }
}

/// <summary>
/// Constructor that takes one argument
/// </summary>
/// <param name="driveInfo">Drive provided by this provider</param>
public AccessDBPSDriveInfo(PSDriveInfo driveInfo)
    : base(driveInfo)
{ }

} // class AccessDBPSDriveInfo

#endregion AccessDBPSDriveInfo

#region DatabaseTableInfo

/// <summary>
/// Contains information specific to the database table.
/// Similar to the DirectoryInfo class.
/// </summary>
public class DatabaseTableInfo
{
    /// <summary>
    /// Row from the "tables" schema
    /// </summary>
    public DataRow Data
    {
        get
        {
            return data;
        }
        set
        {
            data = value;
        }
    }
    private DataRow data;

    /// <summary>
    /// The table name.
    /// </summary>
    public string Name
    {
        get
        {
            return name;
        }
    }
}

```

```

        }
        set
        {
            name = value;
        }
    }
    private String name;

    /// <summary>
    /// The number of rows in the table.
    /// </summary>
    public int RowCount
    {
        get
        {
            return rowCount;
        }
        set
        {
            rowCount = value;
        }
    }
    private int rowCount;

    /// <summary>
    /// The column definitions for the table.
    /// </summary>
    public DataColumnCollection Columns
    {
        get
        {
            return columns;
        }
        set
        {
            columns = value;
        }
    }
    private DataColumnCollection columns;

    /// <summary>
    /// Constructor.
    /// </summary>
    /// <param name="row">The row definition.</param>
    /// <param name="name">The table name.</param>
    /// <param name="rowCount">The number of rows in the table.</param>
    /// <param name="columns">Information on the column tables.</param>
    public DatabaseTableInfo(DataRow row, string name, int rowCount,
                             DataColumnCollection columns)
    {
        Name = name;
        Data = row;
        RowCount = rowCount;
        Columns = columns;
    } // DatabaseTableInfo
}

```

```
 } // class DatabaseTableInfo

#endregion DatabaseTableInfo

#region DatabaseRowInfo

/// <summary>
/// Contains information specific to an individual table row.
/// Analogous to the FileInfo class.
/// </summary>
public class DatabaseRowInfo
{
    /// <summary>
    /// Row data information.
    /// </summary>
    public DataRow Data
    {
        get
        {
            return data;
        }
        set
        {
            data = value;
        }
    }
    private DataRow data;

    /// <summary>
    /// The row index.
    /// </summary>
    public string RowNumber
    {
        get
        {
            return rowNum;
        }
        set
        {
            rowNum = value;
        }
    }
    private string rowNum;

    /// <summary>
    /// Constructor.
    /// </summary>
    /// <param name="row">The row information.</param>
    /// <param name="name">The row index.</param>
    public DatabaseRowInfo(DataRow row, string name)
    {
        RowNumber = name;
        Data = row;
    } // DatabaseRowInfo
} // class DatabaseRowInfo
```

```
#endregion DatabaseRowInfo  
  
#endregion Helper Classes  
}
```

See Also

[System.Management.Automation.Provider.ItemCmdletProvider](#)

[System.Management.Automation.Provider.ContainerCmdletProvider](#)

[System.Management.Automation.Provider.NavigationCmdletProvider](#)

[Designing Your Windows PowerShell Provider](#)

AccessDBProviderSample06

Article • 03/24/2025

This sample shows how to overwrite content methods to support calls to the `Clear-Content`, `Get-Content`, and `Set-Content` cmdlets. These methods should be implemented when the user needs to manage the content of the items in the data store. The provider class in this sample derives from the `System.Management.Automation.Provider.NavigationCmdletProvider` class, and it implements the `System.Management.Automation.Provider.IContentCmdletProvider` interface.

Demonstrates

ⓘ Important

Your provider class will most likely derive from one of the following classes and possibly implement other provider interfaces:

- [System.Management.Automation.Provider.ItemCmdletProvider](#) class. See [AccessDBProviderSample03](#).
- [System.Management.Automation.Provider.ContainerCmdletProvider](#) class. See [AccessDBProviderSample04](#).
- [System.Management.Automation.Provider.NavigationCmdletProvider](#) class.

For more information about choosing which provider class to derive from based on provider features, see [Designing Your Windows PowerShell Provider](#).

This sample demonstrates the following:

- Declaring the `CmdletProvider` attribute.
- Defining a provider class that derives from the `System.Management.Automation.Provider.NavigationCmdletProvider` class and that declares the `System.Management.Automation.Provider.IContentCmdletProvider` interface.
- Overwriting the `System.Management.Automation.Provider.IContentCmdletProvider.ClearContent*` method to change the behavior of the `Clear-Content` cmdlet, allowing the user to remove the content from an item. (This sample does not show how to add dynamic parameters to the `Clear-Content` cmdlet.)

- Overwriting the `System.Management.Automation.Provider.IContentCmdletProvider.GetContentReader*` method to change the behavior of the `Get-Content` cmdlet, allowing the user to retrieve the content of an item. (This sample does not show how to add dynamic parameters to the `Get-Content` cmdlet.).
- Overwriting the `Microsoft.PowerShell.Commands.FileSystemProvider.GetContentWriter*` method to change the behavior of the `Set-Content` cmdlet, allowing the user to update the content of an item. (This sample does not show how to add dynamic parameters to the `Set-Content` cmdlet.)

Example

This sample shows how to overwrite the methods needed to clear, get, and set the content of items in a Microsoft Access data base.

C#

```
using System;
using System.IO;
using System.Data;
using System.Data.Odbc;
using System.Diagnostics;
using System.Collections;
using System.Collections.ObjectModel;
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.Text;
using System.Text.RegularExpressions;
using System.ComponentModel;
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

    /// <summary>
    /// This example implements the content methods.
    /// </summary>
    [CmdletProvider("AccessDB", ProviderCapabilities.None)]
    public class AccessDBProvider : NavigationCmdletProvider,
    IContentCmdletProvider
    {

        #region Drive Manipulation

        /// <summary>
        /// Create a new drive. Create a connection to the database file
        /// </summary>
```

```

and set
    /// the Connection property in the PSDriveInfo.
    /// </summary>
    /// <param name="drive">
    /// Information describing the drive to add.
    /// </param>
    /// <returns>The added drive.</returns>
protected override PSDriveInfo NewDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            null)
        );
    }

    return null;
}

// check if drive root is not null or empty
// and if its an existing file
if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root)
== false))
{
    WriteError(new ErrorRecord(
        new ArgumentException("drive.Root"),
        "NoRoot",
        ErrorCategory.InvalidArgument,
        drive)
    );
}

return null;
}

// create a new drive and create an ODBC connection to the new
drive
AccessDBPSDriveInfo accessDBPSDriveInfo = new
AccessDBPSDriveInfo(drive);

OdbcConnectionStringBuilder builder = new
OdbcConnectionStringBuilder();

builder.Driver = "Microsoft Access Driver (*.mdb)";
builder.Add("DBQ", drive.Root);

OdbcConnection conn = new
OdbcConnection(builder.ConnectionString);
conn.Open();
accessDBPSDriveInfo.Connection = conn;

return accessDBPSDriveInfo;
} // NewDrive

```

```

/// <summary>
/// Removes a drive from the provider.
/// </summary>
/// <param name="drive">The drive to remove.</param>
/// <returns>The drive removed.</returns>
protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            drive)
        );
    }

    return null;
}

// close ODBC connection to the drive
AccessDBPSDriveInfo accessDBPSDriveInfo = drive as
AccessDBPSDriveInfo;

if (accessDBPSDriveInfo == null)
{
    return null;
}
accessDBPSDriveInfo.Connection.Close();

return accessDBPSDriveInfo;
} // RemoveDrive

#endregion Drive Manipulation

#region Item Methods

/// <summary>
/// Retrieves an item using the specified path.
/// </summary>
/// <param name="path">The path to the item to return.</param>
protected override void GetItem(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        WriteItemObject(this.PSDriveInfo, path, true);
        return;
    } // if (PathIsDrive...

    // Get table name and row information from the path and do
    // necessary actions
    string tableName;
    int rowNumber;
}

```

```

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Table)
        {
            DatabaseTableInfo table = GetTable(tableName);
            WriteItemObject(table, path, true);
        }
        else if (type == PathType.Row)
        {
            DatabaseRowInfo row = GetRow(tableName, rowNumber);
            WriteItemObject(row, path, false);
        }
        else
        {
            ThrowTerminatingInvalidOperationException(path);
        }

    } // GetItem

    /// <summary>
    /// Set the content of a row of data specified by the supplied path
    /// parameter.
    /// </summary>
    /// <param name="path">Specifies the path to the row whose columns
    /// will be updated.</param>
    /// <param name="values">Comma separated string of values</param>
protected override void SetItem(string path, object values)
{
    // Get type, table name and row number from the path specified
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type != PathType.Row)
    {
        WriteError(new ErrorRecord(new NotSupportedException(
            "SetNotSupported"), "", 
        ErrorCategory.InvalidOperation, path));

        return;
    }

    // Get in-memory representation of table
    OdbcDataAdapter da = GetAdapterForTable(tableName);

    if (da == null)
    {
        return;
    }
    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);
}

```

```

    if (rowNumber >= table.Rows.Count)
    {
        // The specified row number has to be available. If not
        // NewItem has to be used to add a new row
        throw new ArgumentException("Row specified is not
available");
    } // if (rowNum...

    string[] colValues = (values as string).Split(',');
}

// set the specified row
DataRow row = table.Rows[rowNumber];

for (int i = 0; i < colValues.Length; i++)
{
    row[i] = colValues[i];
}

// Update the table
if (ShouldProcess(path, "SetItem"))
{
    da.Update(ds, tableName);
}

} // SetItem

/// <summary>
/// Test to see if the specified item exists.
/// </summary>
/// <param name="path">The path to the item to verify.</param>
/// <returns>True if the item is found.</returns>
protected override bool ItemExists(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        return true;
    }

    // Obtain type, table name and row number from path
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    DatabaseTableInfo table = GetTable(tableName);

    if (type == PathType.Table)
    {
        // if specified path represents a table then
DatabaseTableInfo
            // object for the same should exist
            if (table != null)

```

```

        {
            return true;
        }
    }
    else if (type == PathType.Row)
    {
        // if specified path represents a row then DatabaseTableInfo
should
        // exist for the table and then specified row number must be
within
        // the maximum row count in the table
        if (table != null && rowNum < table.RowCount)
        {
            return true;
        }
    }

    return false;
}

} // ItemExists

/// <summary>
/// Test to see if the specified path is syntactically valid.
/// </summary>
/// <param name="path">The path to validate.</param>
/// <returns>True if the specified path is valid.</returns>
protected override bool IsValidPath(string path)
{
    bool result = true;

    // check if the path is null or empty
    if (String.IsNullOrEmpty(path))
    {
        result = false;
    }

    // convert all separators in the path to a uniform one
    path = NormalizePath(path);

    // split the path into individual chunks
    string[] pathChunks = path.Split(pathSeparator.ToCharArray());

    foreach (string pathChunk in pathChunks)
    {
        if (pathChunk.Length == 0)
        {
            result = false;
        }
    }
    return result;
} // IsValidPath

#endregion Item Overloads

#region Container Overloads

```

```

/// <summary>
/// Return either the tables in the database or the datarows
/// </summary>
/// <param name="path">The path to the parent</param>
/// <param name="recurse">True to return all child items
recursively.
/// </param>
protected override void GetChildItems(string path, bool recurse)
{
    // If path represented is a drive then the children in the path
are
    // tables. Hence all tables in the drive represented will have
to be
    // returned
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table, path, true);

            // if the specified item exists and recurse has been set
then
            // all child items within it have to be obtained as well
            if (ItemExists(path) && recurse)
            {
                GetChildItems(path + pathSeparator + table.Name,
recurse);
            }
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get the table name, row number and type of path from the
        // path specified
        string tableName;
        int rowCount;

        PathType type = GetNamesFromPath(path, out tableName, out
rowCount);

        if (type == PathType.Table)
        {
            // Obtain all the rows within the table
            foreach (DatabaseRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row, path + pathSeparator +
row.RowNumber,
                                false);
            } // foreach (DatabaseRowInfo...
        }
        else if (type == PathType.Row)
        {
            // In this case the user has directly specified a row,
hence

```

```

        // just give that particular row
        DatabaseRowInfo row = GetRow(tableName, rowNumber);
        WriteItemObject(row, path + pathSeparator +
row.RowNumber,
                        false);
    }
    else
    {
        // In this case, the path specified is not valid
        ThrowTerminatingInvalidPathException(path);
    }
} // else
} // GetChildItems

/// <summary>
/// Return the names of all child items.
/// </summary>
/// <param name="path">The root path.</param>
/// <param name="returnContainers">Not used.</param>
protected override void GetChildNames(string path,
                                         ReturnContainers returnContainers)
{
    // If the path represented is a drive, then the child items are
    // tables. get the names of all the tables in the drive.
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table.Name, path, true);
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get type, table name and row number from path specified
        string tableName;
        int rowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Table)
        {
            // Get all the rows in the table and then write out the
            // row numbers.
            foreach (DatabaseRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row.RowNumber, path, false);
            } // foreach (DatabaseRowInfo...
        }
        else if (type == PathType.Row)
        {
            // In this case the user has directly specified a row,
            hence
            // just give that particular row
            DatabaseRowInfo row = GetRow(tableName, rowNumber);
        }
    }
}

```

```

        WriteItemObject(row.RowNumber, path, false);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }
} // else
} // GetChildNames

/// <summary>
/// Determines if the specified path has child items.
/// </summary>
/// <param name="path">The path to examine.</param>
/// <returns>
/// True if the specified path has child items.
/// </returns>
protected override bool HasChildItems(string path)
{
    if (PathIsDrive(path))
    {
        return true;
    }

    return (ChunkPath(path).Length == 1);
} // HasChildItems

/// <summary>
/// Creates a new item at the specified path.
/// </summary>
///
/// <param name="path">
/// The path to the new item.
/// </param>
///
/// <param name="type">
/// Type for the object to create. "Table" for creating a new table
and
/// "Row" for creating a new row in a table.
/// </param>
///
/// <param name="newValue">
/// Object for creating new instance of a type at the specified
path. For
/// creating a "Table" the object parameter is ignored and for
creating
/// a "Row" the object must be of type string which will contain
comma
/// separated values of the rows to insert.
/// </param>
protected override void NewItem(string path, string type,
                                object newValue)
{
    string tableName;
    int rowNumber;
}

```

```

PathType pt = GetNamesFromPath(path, out tableName, out
rowNumber);

if (pt == PathType.Invalid)
{
    ThrowTerminatingInvalidPathException(path);
}

// Check if type is either "table" or "row", if not throw an
// exception
if (!String.Equals(type, "table",
 StringComparison.OrdinalIgnoreCase)
    && !String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
{
    WriteError(new ErrorRecord
        (new ArgumentException("Type must be
either a table or row"),
         "CannotCreateSpecifiedObject",
         ErrorCategory.InvalidArgument,
         path
        )
    );
}

throw new ArgumentException("This provider can only create
items of type \"table\" or \"row\"");
}

// Path type is the type of path of the container. So if a drive
// is specified, then a table can be created under it and if a
table
// is specified, then a row can be created under it. For the
sake of
// completeness, if a row is specified, then if the row
specified by
// the path does not exist, a new row is created. However, the
row
// number may not match as the row numbers only get incremented
based
// on the number of rows

if (PathIsDrive(path))
{
    if (String.Equals(type, "table",
 StringComparison.OrdinalIgnoreCase))
    {
        // Execute command using ODBC connection to create a
table
        try
        {
            // create the table using an sql statement
            string newItemName = newItemValue.ToString();
            string sql = "create table " + newItemName
                + " (ID INT)";
        }
    }
}

```

```

        // Create the table using the Odbc connection from
the
        // drive.
AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

        if (di == null)
{
    return;
}
OdbcConnection connection = di.Connection;

        if (ShouldProcess(newTableName, "create"))
{
    OdbcCommand cmd = new OdbcCommand(sql,
connection);
    cmd.ExecuteScalar();
}
catch (Exception ex)
{
    WriteError(new ErrorRecord(ex,
"CannotCreateSpecifiedTable",
ErrorCategory.InvalidOperation, path)
);
}
} // if (String...
else if (String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
{
    throw new
ArgumentException("A row cannot be created under a
database, specify a path that represents a Table");
}
}// if (PathIsDrive...
else
{
    if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
    {
        if (rowNumber < 0)
        {
            throw new
ArgumentException("A table cannot be created
within another table, specify a path that represents a database");
        }
        else
        {
            throw new
ArgumentException("A table cannot be created
inside a row, specify a path that represents a database");
        }
    } //if (String.Equals....
// if path specified is a row, create a new row

```

```

        else if (String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
    {
        // The user is required to specify the values to be
        // inserted
        // into the table in a single string separated by commas
        string value = newItemValue as string;

        if (String.IsNullOrEmpty(value))
        {
            throw new
                ArgumentException("Value argument must have
comma separated values of each column in a row");
        }
        string[] rowValues = value.Split(',');
    }

    OdbcDataAdapter da = GetAdapterForTable(tableName);

    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    if (rowValues.Length != table.Columns.Count)
    {
        string message =
String.Format(CultureInfo.CurrentCulture,
                    "The table has {0} columns and
the value specified must have so many comma separated values",
                    table.Columns.Count);

        throw new ArgumentException(message);
    }

    if (!Force && (rowNumber >= 0 && rowNumber <
table.Rows.Count))
    {
        string message =
String.Format(CultureInfo.CurrentCulture,
                    "The row {0} already exists. To
create a new row specify row number as {1}, or specify path to a table, or
use the -Force parameter",
                    rowNumber,
                    table.Rows.Count);

        throw new ArgumentException(message);
    }

    if (rowNumber > table.Rows.Count)
    {
        string message =
String.Format(CultureInfo.CurrentCulture,

```

```

        "To create a new row specify row
number as {0}, or specify path to a table",
                table.Rows.Count);

                throw new ArgumentException(message);
}

// Create a new row and update the row with the input
// provided by the user
DataRow row = table.NewRow();
for (int i = 0; i < rowValues.Length; i++)
{
    row[i] = rowValues[i];
}
table.Rows.Add(row);

if (ShouldProcess(tableName, "update rows"))
{
    // Update the table from memory back to the data
source
    da.Update(ds, tableName);
}

}// else if (String...
}// else ...

} // NewItem

/// <summary>
/// Copies an item at the specified path to the location specified
/// </summary>
///
/// <param name="path">
/// Path of the item to copy
/// </param>
///
/// <param name="copyPath">
/// Path of the item to copy to
/// </param>
///
/// <param name="recurse">
/// Tells the provider to recurse subcontainers when copying
/// </param>
///
protected override void CopyItem(string path, string copyPath, bool
recurse)
{
    string tableName, copyTableName;
    int rowCount, copyRowCount;

    PathType type = GetNamesFromPath(path, out tableName, out
rowCount);
    PathType copyType = GetNamesFromPath(copyPath, out
copyTableName, out copyRowCount);

```

```

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(copyPath);
    }

    // Get the table and the table to copy to
    OdbcDataAdapter da = GetAdapterForTable(tableName);
    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    OdbcDataAdapter cda = GetAdapterForTable(copyTableName);
    if (cda == null)
    {
        return;
    }

    DataSet cds = GetDataSetForTable(cda, copyTableName);
    DataTable copyTable = GetDataTable(cds, copyTableName);

    // if source represents a table
    if (type == PathType.Table)
    {
        // if copyPath does not represent a table
        if (copyType != PathType.Table)
        {
            ArgumentException e = new ArgumentException("Table can
only be copied on to another table location");

            WriteError(new ErrorRecord(e, "PathNotValid",
                ErrorCategory.InvalidArgument, copyPath));

            throw e;
        }
    }

    // if table already exists then force parameter should be
set
    // to force a copy
    if (!Force && GetTable(copyTableName) != null)
    {
        throw new ArgumentException("Specified path already
exists");
    }

    for (int i = 0; i < table.Rows.Count; i++)
    {

```

```

        DataRow row = table.Rows[i];
        DataRow copyRow = copyTable.NewRow();

        copyRow.ItemArray = row.ItemArray;
        copyTable.Rows.Add(copyRow);
    }
} // if (type == ...
// if source represents a row
else
{
    if (copyType == PathType.Row)
    {
        if (!Force && (copyRowNumber < copyTable.Rows.Count))
        {
            throw new ArgumentException("Specified path already
exists.");
        }

        DataRow row = table.Rows[rowNumber];
        DataRow copyRow = null;

        if (copyRowNumber < copyTable.Rows.Count)
        {
            // copy to an existing row
            copyRow = copyTable.Rows[copyRowNumber];
            copyRow.ItemArray = row.ItemArray;
            copyRow[0] = GetNextID(copyTable);
        }
        else if (copyRowNumber == copyTable.Rows.Count)
        {
            // copy to the next row in the table that will
            // be created
            copyRow = copyTable.NewRow();
            copyRow.ItemArray = row.ItemArray;
            copyRow[0] = GetNextID(copyTable);
            copyTable.Rows.Add(copyRow);
        }
        else
        {
            // attempting to copy to a nonexistent row or a row
            // that cannot be created now - throw an exception
            string message =
String.Format(CultureInfo.CurrentCulture,
                         "The item cannot be specified to
the copied row. Specify row number as {0}, or specify a path to the table.",
                         table.Rows.Count);

            throw new ArgumentException(message);
        }
    }
    else
    {
        // destination path specified represents a table,
        // create a new row and copy the item
        DataRow copyRow = copyTable.NewRow();
    }
}

```

```

        copyRow.ItemArray = table.Rows[rowNumber].ItemArray;
        copyRow[0] = GetNextID(copyTable);
        copyTable.Rows.Add(copyRow);
    }
}

if (ShouldProcess(copyTableName, "CopyItems"))
{
    cda.Update(cds, copyTableName);
}

} //CopyItem

/// <summary>
/// Removes (deletes) the item at the specified path
/// </summary>
///
/// <param name="path">
/// The path to the item to remove.
/// </param>
///
/// <param name="recurse">
/// True if all children in a subtree should be removed, false if
only
/// the item at the specified path should be removed. Is applicable
/// only for container (table) items. Its ignored otherwise (even if
/// specified).
/// </param>
///
/// <remarks>
/// There are no elements in this store which are hidden from the
user.
/// Hence this method will not check for the presence of the Force
/// parameter
/// </remarks>
///

protected override void RemoveItem(string path, bool recurse)
{
    string tableName;
    int rowNumber = 0;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        // if recurse flag has been specified, delete all the rows
as well
        if (recurse)
        {
            OdbcDataAdapter da = GetAdapterForTable(tableName);
            if (da == null)
            {
                return;
            }
        }
    }
}

```

```

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        for (int i = 0; i < table.Rows.Count; i++)
        {
            table.Rows[i].Delete();
        }

        if (ShouldProcess(path, "RemoveItem"))
        {
            da.Update(ds, tableName);
            RemoveTable(tableName);
        }
    }//if (recurse...
    else
    {
        // Remove the table
        if (ShouldProcess(path, "RemoveItem"))
        {
            RemoveTable(tableName);
        }
    }
}

else if (type == PathType.Row)
{
    OdbcDataAdapter da = GetAdapterForTable(tableName);
    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    table.Rows[rowNumber].Delete();

    if (ShouldProcess(path, "RemoveItem"))
    {
        da.Update(ds, tableName);
    }
}
else
{
    ThrowTerminatingInvalidOperationException(path);
}

} // RemoveItem

#endif Container Overloads

#region Navigation

/// <summary>
/// Determine if the path specified is that of a container.

```

```

/// </summary>
/// <param name="path">The path to check.</param>
/// <returns>True if the path specifies a container.</returns>
protected override bool IsItemContainer(string path)
{
    if (PathIsDrive(path))
    {
        return true;
    }

    string[] pathChunks = ChunkPath(path);
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        foreach (DatabaseTableInfo ti in GetTables())
        {
            if (string.Equals(ti.Name, tableName,
 StringComparison.OrdinalIgnoreCase))
            {
                return true;
            }
        } // foreach (DatabaseTableInfo...
    } // if (pathChunks...

    return false;
} // IsItemContainer

/// <summary>
/// Get the name of the leaf element in the specified path
/// </summary>
///
/// <param name="path">
/// The full or partial provider specific path
/// </param>
///
/// <returns>
/// The leaf element in the path
/// </returns>
protected override string GetChildName(string path)
{
    if (PathIsDrive(path))
    {
        return path;
    }

    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

```

```

        if (type == PathType.Table)
        {
            return tableName;
        }
        else if (type == PathType.Row)
        {
            return rowNumber.ToString(CultureInfo.CurrentCulture);
        }
        else
        {
            ThrowTerminatingInvalidOperationException(path);
        }

        return null;
    }

    /// <summary>
    /// Removes the child segment of the path and returns the remaining
    /// parent portion
    /// </summary>
    ///
    /// <param name="path">
    /// A full or partial provider specific path. The path may be to an
    /// item that may or may not exist.
    /// </param>
    ///
    /// <param name="root">
    /// The fully qualified path to the root of a drive. This parameter
    /// may be null or empty if a mounted drive is not in use for this
    /// operation. If this parameter is not null or empty the result
    /// of the method should not be a path to a container that is a
    /// parent or in a different tree than the root.
    /// </param>
    ///
    /// <returns></returns>

    protected override string GetParentPath(string path, string root)
    {
        // If root is specified then the path has to contain
        // the root. If not nothing should be returned
        if (!String.IsNullOrEmpty(root))
        {
            if (!path.Contains(root))
            {
                return null;
            }
        }

        return path.Substring(0, path.LastIndexOf(pathSeparator,
StringComparison.OrdinalIgnoreCase));
    }

    /// <summary>
    /// Joins two strings with a provider specific path separator.

```

```

///</summary>
///
///<param name="parent">
/// The parent segment of a path to be joined with the child.
///</param>
///
///<param name="child">
/// The child segment of a path to be joined with the parent.
///</param>
///
///<returns>
/// A string that represents the parent and child segments of the
path
/// joined by a path separator.
///</returns>

protected override string MakePath(string parent, string child)
{
    string result;

    string normalParent = NormalizePath(parent);
    normalParent = RemoveDriveFromPath(normalParent);
    string normalChild = NormalizePath(child);
    normalChild = RemoveDriveFromPath(normalChild);

    if (String.IsNullOrEmpty(normalParent) &&
String.IsNullOrEmpty(normalChild))
    {
        result = String.Empty;
    }
    else if (String.IsNullOrEmpty(normalParent) &&
!String.IsNullOrEmpty(normalChild))
    {
        result = normalChild;
    }
    else if (!String.IsNullOrEmpty(normalParent) &&
String.IsNullOrEmpty(normalChild))
    {
        if (normalParent.EndsWith(pathSeparator,
 StringComparison.OrdinalIgnoreCase))
        {
            result = normalParent;
        }
        else
        {
            result = normalParent + pathSeparator;
        }
    } // else if (!String...
    else
    {
        if (!normalParent.Equals(String.Empty,
 StringComparison.OrdinalIgnoreCase) &&
            !normalParent.EndsWith(pathSeparator,
 StringComparison.OrdinalIgnoreCase))
        {

```

```

                result = normalParent + pathSeparator;
            }
            else
            {
                result = normalParent;
            }

            if (normalChild.StartsWith(pathSeparator,
StringComparison.OrdinalIgnoreCase))
            {
                result += normalChild.Substring(1);
            }
            else
            {
                result += normalChild;
            }
        } // else

        return result;
    } // MakePath

    /// <summary>
    /// Normalizes the path that was passed in and returns the
normalized
    /// path as a relative path to the basePath that was passed.
    /// </summary>
    ///
    /// <param name="path">
    /// A fully qualified provider specific path to an item. The item
    /// should exist or the provider should write out an error.
    /// </param>
    ///
    /// <param name="basepath">
    /// The path that the return value should be relative to.
    /// </param>
    ///
    /// <returns>
    /// A normalized path that is relative to the basePath that was
    /// passed. The provider should parse the path parameter, normalize
    /// the path, and then return the normalized path relative to the
    /// basePath.
    /// </returns>

    protected override string NormalizeRelativePath(string path,
                                                string
basepath)
{
    // Normalize the paths first
    string normalPath = NormalizePath(path);
    normalPath = RemoveDriveFromPath(normalPath);
    string normalBasePath = NormalizePath(basepath);
    normalBasePath = RemoveDriveFromPath(normalBasePath);

    if (String.IsNullOrEmpty(normalBasePath))
    {

```

```

        return normalPath;
    }
    else
    {
        if (!normalPath.Contains(normalbasePath))
        {
            return null;
        }

        return normalPath.Substring(normalbasePath.Length +
pathSeparator.Length);
    }
}

/// <summary>
/// Moves the item specified by the path to the specified
destination
/// </summary>
///
/// <param name="path">
/// The path to the item to be moved
/// </param>
///
/// <param name="destination">
/// The path of the destination container
/// </param>

protected override void MoveItem(string path, string destination)
{
    // Get type, table name and rowNumber from the path
    string tableName, destTableName;
    int rowNumber, destRowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    PathType destType = GetNamesFromPath(destination, out
destTableName,
                                         out destRowNumber);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }

    if (destType == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(destination);
    }

    if (type == PathType.Table)
    {
        ArgumentException e = new ArgumentException("Move not
supported for tables");

```

```

        WriteError(new ErrorRecord(e, "MoveNotSupported",
            ErrorCategory.InvalidArgument, path));

        throw e;
    }
    else
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);
        if (da == null)
        {
            return;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        OdbcDataAdapter dda = GetAdapterForTable(destTableName);
        if (dda == null)
        {
            return;
        }

        DataSet dds = GetDataSetForTable(dda, destTableName);
        DataTable destTable = GetDataTable(dds, destTableName);
        DataRow row = table.Rows[rowNumber];

        if (destType == PathType.Table)
        {
            DataRow destRow = destTable.NewRow();

            destRow.ItemArray = row.ItemArray;
        }
        else
        {
            DataRow destRow = destTable.Rows[destRowNumber];

            destRow.ItemArray = row.ItemArray;
        }

        // Update the changes
        if (ShouldProcess(path, "MoveItem"))
        {
            WriteItemObject(row, path, false);
            dda.Update(dds, destTableName);
        }
    }
}

#endregion Navigation

#region Helper Methods

/// <summary>
/// Checks if a given path is actually a drive name.
/// </summary>

```

```

/// <param name="path">The path to check.</param>
/// <returns>
/// True if the path given represents a drive, false otherwise.
/// </returns>
private bool PathIsDrive(string path)
{
    // Remove the drive name and first path separator. If the
    // path is reduced to nothing, it is a drive. Also if its
    // just a drive then there wont be any path separators
    if (String.IsNullOrEmpty(
        path.Replace(this.PSDriveInfo.Root, ""))
        ||
        String.IsNullOrEmpty(
            path.Replace(this.PSDriveInfo.Root + pathSeparator,
            "")))
    {
        return true;
    }
    else
    {
        return false;
    }
} // PathIsDrive

/// <summary>
/// Breaks up the path into individual elements.
/// </summary>
/// <param name="path">The path to split.</param>
/// <returns>An array of path segments.</returns>
private string[] ChunkPath(string path)
{
    // Normalize the path before splitting
    string normalPath = NormalizePath(path);

    // Return the path with the drive name and first path
    // separator character removed, split by the path separator.
    string pathNoDrive = normalPath.Replace(this.PSDriveInfo.Root
        + pathSeparator, "");

    return pathNoDrive.Split(pathSeparator.ToCharArray());
} // ChunkPath

/// <summary>
/// Adapts the path, making sure the correct path separator
/// character is used.
/// </summary>
/// <param name="path"></param>
/// <returns></returns>
private string NormalizePath(string path)
{
    string result = path;

    if (!String.IsNullOrEmpty(path))
    {

```

```

        result = path.Replace("/", pathSeparator);
    }

    return result;
} // NormalizePath

/// <summary>
/// Ensures that the drive is removed from the specified path
/// </summary>
///
/// <param name="path">Path from which drive needs to be
removed</param>
/// <returns>Path with drive information removed</returns>
private string RemoveDriveFromPath(string path)
{
    string result = path;
    string root;

    if (this.PSDriveInfo == null)
    {
        root = String.Empty;
    }
    else
    {
        root = this.PSDriveInfo.Root;
    }

    if (result == null)
    {
        result = String.Empty;
    }

    if (result.Contains(root))
    {
        result = result.Substring(result.IndexOf(root,
StringComparison.OrdinalIgnoreCase) + root.Length);
    }

    return result;
}

/// <summary>
/// Chunks the path and returns the table name and the row number
/// from the path
/// </summary>
/// <param name="path">Path to chunk and obtain information</param>
/// <param name="tableName">Name of the table as represented in the
/// path</param>
/// <param name="rowNumber">Row number obtained from the
path</param>
/// <returns>what the path represents</returns>
public PathType GetNamesFromPath(string path, out string tableName,
out int rowNumber)
{
    PathType retVal = PathType.Invalid;

```

```

rowNumber = -1;
tableName = null;

// Check if the path specified is a drive
if (PathIsDrive(path))
{
    return PathType.Database;
}

// chunk the path into parts
string[] pathChunks = ChunkPath(path);

switch (pathChunks.Length)
{
    case 1:
    {
        string name = pathChunks[0];

        if (TableNameIsValid(name))
        {
            tableName = name;
            retVal = PathType.Table;
        }
    }
    break;

    case 2:
    {
        string name = pathChunks[0];

        if (TableNameIsValid(name))
        {
            tableName = name;
        }

        int number = SafeConvertRowNumber(pathChunks[1]);

        if (number >= 0)
        {
            rowNumber = number;
            retVal = PathType.Row;
        }
        else
        {
            WriteError(new ErrorRecord(
                new ArgumentException("Row number is not
valid"),
                "RowNumberNotValid",
                ErrorCategory.InvalidArgument,
                path));
        }
    }
    break;

    default:
}

```

```

        {
            WriteError(new ErrorRecord(
                new ArgumentException("The path supplied has too
many segments"),
                "PathNotValid",
                ErrorCategory.InvalidArgument,
                path)));
        }
        break;
    } // switch(pathChunks...

    return retVal;
} // GetNamesFromPath

/// <summary>
/// Throws an argument exception stating that the specified path
does
/// not represent either a table or a row
/// </summary>
/// <param name="path">path which is invalid</param>
private void ThrowTerminatingInvalidPathException(string path)
{
    StringBuilder message = new StringBuilder("Path must represent
either a table or a row :");
    message.Append(path);

    throw new ArgumentException(message.ToString());
}

/// <summary>
/// Retrieve the list of tables from the database.
/// </summary>
/// <returns>
/// Collection of DatabaseTableInfo objects, each object
representing
/// information about one database table
/// </returns>
internal Collection<DatabaseTableInfo> GetTables()
{
    Collection<DatabaseTableInfo> results =
        new Collection<DatabaseTableInfo>();

    // using ODBC connection to the database and get the schema of
tables
    AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

    if (di == null)
    {
        return null;
    }

    OdbcConnection connection = di.Connection;
    DataTable dt = connection.GetSchema("Tables");
    int count;
}

```

```

        // iterate through all rows in the schema and create
DatabaseTableInfo
            // objects which represents a table
foreach (DataRow dr in dt.Rows)
{
    String tableName = dr["TABLE_NAME"] as String;
    DataColumnCollection columns = null;

        // find the number of rows in the table
try
{
    String cmd = "Select count(*) from \\" + tableName +
"\\"";
    OdbcCommand command = new OdbcCommand(cmd, connection);

    count = (Int32)command.ExecuteScalar();
}
catch
{
    count = 0;
}

        // create DatabaseTableInfo object representing the table
DatabaseTableInfo table =
    new DatabaseTableInfo(dr, tableName, count,
columns);

    results.Add(table);
} // foreach (DataRow...

        return results;
} // GetTables

/// <summary>
/// Return row information from a specified table.
/// </summary>
/// <param name="tableName">The name of the database table from
/// which to retrieve rows.</param>
/// <returns>Collection of row information objects.</returns>
public Collection<DatabaseRowInfo> GetRows(string tableName)
{
    Collection<DatabaseRowInfo> results =
        new Collection<DatabaseRowInfo>();

    // Obtain rows in the table and add it to the collection
try
{
    OdbcDataAdapter da = GetAdapterForTable(tableName);

    if (da == null)
    {
        return null;
    }
}

```

```

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        int i = 0;
        foreach (DataRow row in table.Rows)
        {
            results.Add(new DatabaseRowInfo(row,
i.ToString(CultureInfo.CurrentCulture)));
            i++;
        } // foreach (DataRow...
    }
    catch (Exception e)
    {
        WriteError(new ErrorRecord(e, "CannotAccessSpecifiedRows",
            ErrorCategory.InvalidOperation, tableName));
    }

    return results;
} // GetRows

/// <summary>
/// Retrieve information about a single table.
/// </summary>
/// <param name="tableName">The table for which to retrieve
/// data.</param>
/// <returns>Table information.</returns>
private DatabaseTableInfo GetTable(string tableName)
{
    foreach (DatabaseTableInfo table in GetTables())
    {
        if (String.Equals(tableName, table.Name,
StringComparison.OrdinalIgnoreCase))
        {
            return table;
        }
    }

    return null;
} // GetTable

/// <summary>
/// Removes the specified table from the database
/// </summary>
/// <param name="tableName">Name of the table to remove</param>
private void RemoveTable(string tableName)
{
    // validate if tablename is valid and if table is present
    if (String.IsNullOrEmpty(tableName) ||
!TableNameIsValid(tableName) || !TableIsPresent(tableName))
    {
        return;
    }

    // Execute command using ODBC connection to remove a table
}

```

```

    try
    {
        // delete the table using an sql statement
        string sql = "drop table " + tableName;

        AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

        if (di == null)
        {
            return;
        }
        OdbcConnection connection = di.Connection;

        OdbcCommand cmd = new OdbcCommand(sql, connection);
        cmd.ExecuteScalar();
    }
    catch (Exception ex)
    {
        WriteError(new ErrorRecord(ex, "CannotRemoveSpecifiedTable",
            ErrorCategory.InvalidOperation, null)
        );
    }
}

} // RemoveTable

/// <summary>
/// Obtain a data adapter for the specified Table
/// </summary>
/// <param name="tableName">Name of the table to obtain the
/// adapter for</param>
/// <returns>Adapter object for the specified table</returns>
/// <remarks>An adapter serves as a bridge between a DataSet (in
memory
/// representation of table) and the data source</remarks>
internal OdbcDataAdapter GetAdapterForTable(string tableName)
{
    OdbcDataAdapter da = null;
    AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

    if (di == null || !TableNameIsValid(tableName) ||
!TableIsPresent(tableName))
    {
        return null;
    }

    OdbcConnection connection = di.Connection;

    try
    {
        // Create a odbc data adpater. This can be sued to update
the
        // data source with the records that will be created here
        // using data sets

```

```

        string sql = "Select * from " + tableName;
        da = new OdbcDataAdapter(new OdbcCommand(sql, connection));

        // Create a odbc command builder object. This will create
sql
        // commands automatically for a single table, thus
        // eliminating the need to create new sql statements for
        // every operation to be done.
        OdbcCommandBuilder cmd = new OdbcCommandBuilder(da);

        // Set the delete cmd for the table here
        sql = "Delete from " + tableName + " where ID = ?";
        da.DeleteCommand = new OdbcCommand(sql, connection);

        // Specify a DeleteCommand parameter based on the "ID"
        // column
        da.DeleteCommand.Parameters.Add(new OdbcParameter());
        da.DeleteCommand.Parameters[0].SourceColumn = "ID";

        // Create an InsertCommand based on the sql string
        // Insert into "tablename" values (?,?,?,?) where
        // ? represents a column in the table. Note that
        // the number of ? will be equal to the number of
        // columns
        DataSet ds = new DataSet();
        ds.Locale = CultureInfo.InvariantCulture;

        da.FillSchema(ds, SchemaType.Source);

        sql = "Insert into " + tableName + " values ( ";
        for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
        {
            sql += "?, ";
        }
        sql = sql.Substring(0, sql.Length - 2);
        sql += ")";
        da.InsertCommand = new OdbcCommand(sql, connection);

        // Create parameters for the InsertCommand based on the
        // captions of each column
        for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
        {
            da.InsertCommand.Parameters.Add(new OdbcParameter());
            da.InsertCommand.Parameters[i].SourceColumn =
                ds.Tables["Table"].Columns[i].Caption;
        }

        // Open the connection if its not already open
        if (connection.State != ConnectionState.Open)
        {
            connection.Open();
        }
    }
    catch (Exception e)

```

```

        {
            WriteError(new ErrorRecord(e, "CannotAccessSpecifiedTable",
                ErrorCategory.InvalidOperation, tableName));
        }

        return da;
    } // GetAdapterForTable

    /// <summary>
    /// Gets the DataSet (in memory representation) for the table
    /// for the specified adapter
    /// </summary>
    /// <param name="adapter">Adapter to be used for obtaining
    /// the table</param>
    /// <param name="tableName">Name of the table for which a
    /// DataSet is required</param>
    /// <returns>The DataSet with the filled in schema</returns>
    internal DataSet GetDataSetForTable(OdbcDataAdapter adapter, string
tableName)
{
    Debug.Assert(adapter != null);

    // Create a dataset object which will provide an in-memory
    // representation of the data being worked upon in the
    // data source.
    DataSet ds = new DataSet();

    // Create a table named "Table" which will contain the same
    // schema as in the data source.
    //adapter.FillSchema(ds, SchemaType.Source);
    adapter.Fill(ds, tableName);
    ds.Locale = CultureInfo.InvariantCulture;

    return ds;
} //GetDataSetForTable

    /// <summary>
    /// Get the DataTable object which can be used to operate on
    /// for the specified table in the data source
    /// </summary>
    /// <param name="ds">DataSet object which contains the tables
    /// schema</param>
    /// <param name="tableName">Name of the table</param>
    /// <returns>Corresponding DataTable object representing
    /// the table</returns>
    ///
    internal DataTable GetDataTable(DataSet ds, string tableName)
{
    Debug.Assert(ds != null);
    Debug.Assert(tableName != null);

    DataTable table = ds.Tables[tableName];
    table.Locale = CultureInfo.InvariantCulture;

    return table;
}

```

```

    } // GetDataTable

    /// <summary>
    /// Retrieves a single row from the named table.
    /// </summary>
    /// <param name="tableName">The table that contains the
    /// numbered row.</param>
    /// <param name="row">The index of the row to return.</param>
    /// <returns>The specified table row.</returns>
    private DatabaseRowInfo GetRow(string tableName, int row)
    {
        Collection<DatabaseRowInfo> di = GetRows(tableName);

        // if the row is invalid write an appropriate error else return
        the
        // corresponding row information
        if (row < di.Count && row >= 0)
        {
            return di[row];
        }
        else
        {
            WriteError(new ErrorRecord(
                new ItemNotFoundException(),
                "RowNotFound",
                ErrorCategory.ObjectNotFound,
                row.ToString(CultureInfo.CurrentCulture))
            );
        }

        return null;
    } // GetRow

    /// <summary>
    /// Method to safely convert a string representation of a row number
    /// into its Int32 equivalent
    /// </summary>
    /// <param name="rowNumberAsStr">String representation of the row
    /// number</param>
    /// <remarks>If there is an exception, -1 is returned</remarks>
    private int SafeConvertRowNumber(string rowNumberAsStr)
    {
        int rowNumber = -1;
        try
        {
            rowNumber = Convert.ToInt32(rowNumberAsStr,
CultureInfo.CurrentCulture);
        }
        catch (FormatException fe)
        {
            WriteError(new ErrorRecord(fe, "RowStringFormatNotValid",
                ErrorCategory.InvalidData, rowNumberAsStr));
        }
        catch (OverflowException oe)
        {

```

```

                WriteError(new ErrorRecord(oe,
"RowStringConversionToNumberFailed",
                    ErrorCategory.InvalidData, rowNumberAsStr));
            }

            return rowNumber;
        } // 1

        /// <summary>
        /// Check if a table name is valid
        /// </summary>
        /// <param name="tableName">Table name to validate</param>
        /// <remarks>Helps to check for SQL injection attacks</remarks>
        private bool TableNameIsValid(string tableName)
        {
            Regex exp = new Regex(pattern, RegexOptions.Compiled |
RegexOptions.IgnoreCase);

            if (exp.IsMatch(tableName))
            {
                return true;
            }
            WriteError(new ErrorRecord(
                new ArgumentException("Table name not valid"),
"TableNameNotValid",
                ErrorCategory.InvalidArgument, tableName));
            return false;
        } // TableNameIsValid

        /// <summary>
        /// Checks to see if the specified table is present in the
        /// database
        /// </summary>
        /// <param name="tableName">Name of the table to check</param>
        /// <returns>true, if table is present, false otherwise</returns>
        private bool TableIsPresent(string tableName)
        {
            // using ODBC connection to the database and get the schema of
tables
            AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;
            if (di == null)
            {
                return false;
            }

            OdbcConnection connection = di.Connection;
            DataTable dt = connection.GetSchema("Tables");

            // check if the specified tableName is available
            // in the list of tables present in the database
            foreach (DataRow dr in dt.Rows)
            {
                string name = dr["TABLE_NAME"] as string;
                if (name.Equals(tableName,

```

```

        StringComparison.OrdinalIgnoreCase))
    {
        return true;
    }
}

WriteError(new ErrorRecord(
    new ArgumentException("Specified Table is not present in
database"), "TableNotAvailable",
    ErrorCategory.InvalidArgument, tableName));

return false;
}// TableIsPresent

/// <summary>
/// Gets the next available ID in the table
/// </summary>
/// <param name="table">DataTable object representing the table to
/// search for ID</param>
/// <returns>next available id</returns>
private int GetNextID(DataTable table)
{
    int big = 0;

    for (int i = 0; i < table.Rows.Count; i++)
    {
        DataRow row = table.Rows[i];

        int id = (int)row["ID"];

        if (big < id)
        {
            big = id;
        }
    }

    big++;
    return big;
}
#endregion Helper Methods

#region Content Methods

/// <summary>
/// Clear the contents at the specified location. In this case,
/// clearing
/// the item amounts to clearing a row
/// </summary>
/// <param name="path">The path to the content to clear.</param>
public void ClearContent(string path)
{
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out

```

```

rowNumber);

    if (type != PathType.Table)
    {
        WriteError(new ErrorRecord(
            new InvalidOperationException("Operation not supported.
Content can be cleared only for table"),
            "NotValidRow", ErrorCategory.InvalidArgument,
            path));
        return;
    }

    OdbcDataAdapter da = GetAdapterForTable(tableName);

    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    // Clear contents at the specified location
    for (int i = 0; i < table.Rows.Count; i++)
    {
        table.Rows[i].Delete();
    }

    if (ShouldProcess(path, "ClearContent"))
    {
        da.Update(ds, tableName);
    }

} // ClearContent

/// <summary>
/// Not implemented.
/// </summary>
/// <param name="path"></param>
/// <returns></returns>
public object ClearContentDynamicParameters(string path)
{
    return null;
}

/// <summary>
/// Get a reader at the path specified.
/// </summary>
/// <param name="path">The path from which to read.</param>
/// <returns>A content reader used to read the data.</returns>
public IContentReader GetContentReader(string path)
{
    string tableName;
    int rowNumber;
}

```

```

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Invalid)
        {
            ThrowTerminatingInvalidPathException(path);
        }
        else if (type == PathType.Row)
        {
            throw new InvalidOperationException("contents can be obtained
only for tables");
        }

        return new AccessDBContentReader(path, this);
    } // GetContentReader

    /// <summary>
    /// Not implemented.
    /// </summary>
    /// <param name="path"></param>
    /// <returns></returns>
    public object GetContentReaderDynamicParameters(string path)
    {
        return null;
    }

    /// <summary>
    /// Get an object used to write content.
    /// </summary>
    /// <param name="path">The root path at which to write.</param>
    /// <returns>A content writer for writing.</returns>
    public IContentWriter GetContentWriter(string path)
    {
        string tableName;
        int rowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Invalid)
        {
            ThrowTerminatingInvalidPathException(path);
        }
        else if (type == PathType.Row)
        {
            throw new InvalidOperationException("contents can be added
only to tables");
        }

        return new AccessDBContentWriter(path, this);
    }

    /// <summary>
    /// Not implemented.
    /// </summary>

```

```

    ///<param name="path"></param>
    ///<returns></returns>
    public object GetContentWriterDynamicParameters(string path)
    {
        return null;
    }

#endregion Content Methods

#region Private Properties

private string pathSeparator = "\\";
private static string pattern = @"^+[a-z]+[0-9]*_*$";

#endregion Private Properties

} // AccessDBProvider

#endregion AccessDBProvider

#region Helper Classes

#region Public Enumerations

///<summary>
/// Type of item represented by the path
///</summary>
public enum PathType
{
    ///<summary>
    /// Represents a database
    ///</summary>
    Database,
    ///<summary>
    /// Represents a table
    ///</summary>
    Table,
    ///<summary>
    /// Represents a row
    ///</summary>
    Row,
    ///<summary>
    /// Represents an invalid path
    ///</summary>
    Invalid
};

#endregion Public Enumerations

#region AccessDBPSDriveInfo

///<summary>
/// Any state associated with the drive should be held here.
/// In this case, it's the connection to the database.
///</summary>

```

```

internal class AccessDBPSDriveInfo : PSDriveInfo
{
    private OdbcConnection connection;

    /// <summary>
    /// ODBC connection information.
    /// </summary>
    public OdbcConnection Connection
    {
        get { return connection; }
        set { connection = value; }
    }

    /// <summary>
    /// Constructor that takes one argument
    /// </summary>
    /// <param name="driveInfo">Drive provided by this provider</param>
    public AccessDBPSDriveInfo(PSDriveInfo driveInfo)
        : base(driveInfo)
    {
    }

} // class AccessDBPSDriveInfo

#endregion AccessDBPSDriveInfo

#region DatabaseTableInfo

/// <summary>
/// Contains information specific to the database table.
/// Similar to the DirectoryInfo class.
/// </summary>
public class DatabaseTableInfo
{
    /// <summary>
    /// Row from the "tables" schema
    /// </summary>
    public DataRow Data
    {
        get
        {
            return data;
        }
        set
        {
            data = value;
        }
    }
    private DataRow data;

    /// <summary>
    /// The table name.
    /// </summary>
    public string Name
    {
        get
    }
}

```

```

    {
        return name;
    }
    set
    {
        name = value;
    }
}
private String name;

/// <summary>
/// The number of rows in the table.
/// </summary>
public int RowCount
{
    get
    {
        return rowCount;
    }
    set
    {
        rowCount = value;
    }
}
private int rowCount;

/// <summary>
/// The column definitions for the table.
/// </summary>
public DataColumnCollection Columns
{
    get
    {
        return columns;
    }
    set
    {
        columns = value;
    }
}
private DataColumnCollection columns;

/// <summary>
/// Constructor.
/// </summary>
/// <param name="row">The row definition.</param>
/// <param name="name">The table name.</param>
/// <param name="rowCount">The number of rows in the table.</param>
/// <param name="columns">Information on the column tables.</param>
public DatabaseTableInfo(DataRow row, string name, int rowCount,
                         DataColumnCollection columns)
{
    Name = name;
    Data = row;
    RowCount = rowCount;
}

```

```
        Columns = columns;
    } // DatabaseTableInfo
} // class DatabaseTableInfo

#endregion DatabaseTableInfo

#region DatabaseRowInfo

/// <summary>
/// Contains information specific to an individual table row.
/// Analogous to the FileInfo class.
/// </summary>
public class DatabaseRowInfo
{
    /// <summary>
    /// Row data information.
    /// </summary>
    public DataRow Data
    {
        get
        {
            return data;
        }
        set
        {
            data = value;
        }
    }
    private DataRow data;

    /// <summary>
    /// The row index.
    /// </summary>
    public string RowNumber
    {
        get
        {
            return rowNum;
        }
        set
        {
            rowNum = value;
        }
    }
    private string rowNum;

    /// <summary>
    /// Constructor.
    /// </summary>
    /// <param name="row">The row information.</param>
    /// <param name="name">The row index.</param>
    public DatabaseRowInfo(DataRow row, string name)
    {
        RowNumber = name;
        Data = row;
    }
}
```

```

        } // DatabaseRowInfo
    } // class DatabaseRowInfo

#endregion DatabaseRowInfo

#region AccessDBContentReader

/// <summary>
/// Content reader used to retrieve data from this provider.
/// </summary>
public class AccessDBContentReader : IContentReader
{
    // A provider instance is required so as to get "content"
    private AccessDBProvider provider;
    private string path;
    private long currentOffset;

    internal AccessDBContentReader(string path, AccessDBProvider provider)
    {
        this.path = path;
        this.provider = provider;
    }

    /// <summary>
    /// Read the specified number of rows from the source.
    /// </summary>
    /// <param name="readCount">The number of items to
    /// </param>
    /// <returns>An array of elements read.</returns>
    public IList Read(long readCount)
    {
        // Read the number of rows specified by readCount and increment
        // offset
        string tableName;
        int rowNumber;
        PathType type = provider.GetNamesFromPath(path, out tableName,
out rowNumber);

        Collection<DatabaseRowInfo> rows =
            provider.GetRows(tableName);
        Collection<DataRow> results = new Collection<DataRow>();

        if (currentOffset < 0 || currentOffset >= rows.Count)
        {
            return null;
        }

        int rowsRead = 0;

        while (rowsRead < readCount && currentOffset < rows.Count)
        {
            results.Add(rows[(int)currentOffset].Data);
            rowsRead++;
            currentOffset++;
        }
    }
}

```

```

        }

        return results;
    } // Read

    /// <summary>
    /// Moves the content reader specified number of rows from the
    /// origin
    /// </summary>
    /// <param name="offset">Number of rows to offset</param>
    /// <param name="origin">Starting row from which to offset</param>
    public void Seek(long offset, System.IO.SeekOrigin origin)
    {
        // get the number of rows in the table which will help in
        // calculating current position
        string tableName;
        int rowNumber;

        PathType type = provider.GetNamesFromPath(path, out tableName,
out rowNumber);

        if (type == PathType.Invalid)
        {
            throw new ArgumentException("Path specified must represent a
table or a row :" + path);
        }

        if (type == PathType.Table)
        {
            Collection<DatabaseRowInfo> rows =
provider.GetRows(tableName);

            int numRows = rows.Count;

            if (offset > rows.Count)
            {
                throw new
                    ArgumentException(
                        "Offset cannot be greater than the number of
rows available"
                    );
            }

            if (origin == System.IO.SeekOrigin.Begin)
            {
                // starting from Beginning with an index 0, the current
offset
                // has to be advanced to offset - 1
                currentOffset = offset - 1;
            }
            else if (origin == System.IO.SeekOrigin.End)
            {
                // starting from the end which is numRows - 1, the
current
                // offset is so much less than numRows - 1
            }
        }
    }
}

```

```

        currentOffset = numRows - 1 - offset;
    }
    else
    {
        // calculate from the previous value of current offset
        // advancing forward always
        currentOffset += offset;
    }
} // if (type...
else
{
    // for row, the offset will always be set to 0
    currentOffset = 0;
}

} // Seek

/// <summary>
/// Closes the content reader, so all members are reset
/// </summary>
public void Close()
{
    Dispose();
} // Close

/// <summary>
/// Dispose any resources being used
/// </summary>
public void Dispose()
{
    Seek(0, System.IO.SeekOrigin.Begin);

    GC.SuppressFinalize(this);
} // Dispose
} // AccessDBContentReader

#endregion AccessDBContentReader

#region AccessDBContentWriter

/// <summary>
/// Content writer used to write data in this provider.
/// </summary>
public class AccessDBContentWriter : IContentWriter
{
    // A provider instance is required so as to get "content"
    private AccessDBProvider provider;
    private string path;
    private long currentOffset;

    internal AccessDBContentWriter(string path, AccessDBProvider
provider)
    {
        this.path = path;
        this.provider = provider;
    }
}

```

```
}

/// <summary>
/// Write the specified row contents in the source
/// </summary>
/// <param name="content"> The contents to be written to the source.
/// </param>
/// <returns>An array of elements which were successfully written to
/// the source</returns>
///
public IList Write(IList content)
{
    if (content == null)
    {
        return null;
    }

    // Get the total number of rows currently available it will
    // determine how much to overwrite and how much to append at
    // the end
    string tableName;
    int rowNumber;
    PathType type = provider.GetNamesFromPath(path, out tableName,
out rowNumber);

    if (type == PathType.Table)
    {
        OdbcDataAdapter da = provider.GetAdapterForTable(tableName);
        if (da == null)
        {
            return null;
        }

        DataSet ds = provider.GetDataSetForTable(da, tableName);
        DataTable table = provider.GetDataTable(ds, tableName);

        string[] colValues = (content[0] as string).Split(',');
        // set the specified row
        DataRow row = table.NewRow();

        for (int i = 0; i < colValues.Length; i++)
        {
            if (!String.IsNullOrEmpty(colValues[i]))
            {
                row[i] = colValues[i];
            }
        }

        //table.Rows.InsertAt(row, rowNumber);
        // Update the table
        table.Rows.Add(row);
        da.Update(ds, tableName);
    }
}
```

```

        else
        {
            throw new InvalidOperationException("Operation not
supported. Content can be added only for tables");
        }

        return null;
    } // Write

    /// <summary>
    /// Moves the content reader specified number of rows from the
    /// origin
    /// </summary>
    /// <param name="offset">Number of rows to offset</param>
    /// <param name="origin">Starting row from which to offset</param>
    public void Seek(long offset, System.IO.SeekOrigin origin)
    {
        // get the number of rows in the table which will help in
        // calculating current position
        string tableName;
        int rowNumber;

        PathType type = provider.GetNamesFromPath(path, out tableName,
out rowNumber);

        if (type == PathType.Invalid)
        {
            throw new ArgumentException("Path specified should represent
either a table or a row : " + path);
        }

        Collection<DatabaseRowInfo> rows =
            provider.GetRows(tableName);

        int numRows = rows.Count;

        if (offset > rows.Count)
        {
            throw new
                ArgumentException(
                    "Offset cannot be greater than the number of rows
available"
                );
        }

        if (origin == System.IO.SeekOrigin.Begin)
        {
            // starting from Beginning with an index 0, the current
offset
            // has to be advanced to offset - 1
            currentOffset = offset - 1;
        }
        else if (origin == System.IO.SeekOrigin.End)
        {
            // starting from the end which is numRows - 1, the current

```

```

        // offset is so much less than numRows - 1
        currentOffset = numRows - 1 - offset;
    }
    else
    {
        // calculate from the previous value of current offset
        // advancing forward always
        currentOffset += offset;
    }

} // Seek

/// <summary>
/// Closes the content reader, so all members are reset
/// </summary>
public void Close()
{
    Dispose();
} // Close

/// <summary>
/// Dispose any resources being used
/// </summary>
public void Dispose()
{
    Seek(0, System.IO.SeekOrigin.Begin);

    GC.SuppressFinalize(this);
} // Dispose
} // AccessDBContentWriter

#endregion AccessDBContentWriter

#region Helper Classes
} // namespace Microsoft.Samples.PowerShell.Providers

```

See Also

[System.Management.Automation.Provider.ItemCmdletProvider](#)

[System.Management.Automation.Provider.ContainerCmdletProvider](#)

[System.Management.Automation.Provider.NavigationCmdletProvider](#)

[Designing Your Windows PowerShell Provider](#)

Windows PowerShell Host Quickstart

Article • 12/18/2023

To host Windows PowerShell in your application, you use the [System.Management.Automation.PowerShell](#) class. This class provides methods that create a pipeline of commands and then execute those commands in a runspace. The simplest way to create a host application is to use the default runspace. The default runspace contains all of the core Windows PowerShell commands. If you want your application to expose only a subset of the Windows PowerShell commands, you must create a custom runspace.

Using the default runspace

To start, we'll use the default runspace, and use the methods of the [System.Management.Automation.PowerShell](#) class to add commands, parameters, statements, and scripts to a pipeline.

AddCommand

You use the [System.Management.Automation.PowerShell.AddCommand](#) method to add commands to the pipeline. For example, suppose you want to get the list of running processes on the machine. The way to run this command is as follows.

1. Create a [System.Management.Automation.PowerShell](#) object.

```
C#  
  
PowerShell ps = PowerShell.Create();
```

2. Add the command that you want to execute.

```
C#  
  
ps.AddCommand("Get-Process");
```

3. Invoke the command.

```
C#  
  
ps.Invoke();
```

If you call the `AddCommand` method more than once before you call the `System.Management.Automation.PowerShell.Invoke` method, the result of the first command is piped to the second, and so on. If you do not want to pipe the result of a previous command to a command, add it by calling the `System.Management.Automation.PowerShell.AddStatement` instead.

AddParameter

The previous example executes a single command without any parameters. You can add parameters to the command by using the `System.Management.Automation.PSCmdlet.AddParameter` method. For example, the following code gets a list of all of the processes that are named `powershell` running on the machine.

C#

```
PowerShell.Create().AddCommand("Get-Process")
    .AddParameter("Name", "powershell")
    .Invoke();
```

You can add additional parameters by calling the `AddParameter` method repeatedly.

C#

```
PowerShell.Create().AddCommand("Get-ChildItem")
    .AddParameter("Path", @"C:\Windows")
    .AddParameter("Filter", "*.*")
    .Invoke();
```

You can also add a dictionary of parameter names and values by calling the `System.Management.Automation.PowerShell.AddParameters` method.

C#

```
IDictionary parameters = new Dictionary<String, String>();
parameters.Add("Path", @"C:\Windows");
parameters.Add("Filter", "*.*");

PowerShell.Create().AddCommand("Get-Process")
    .AddParameters(parameters)
    .Invoke()
```

AddStatement

You can simulate batching by using the [System.Management.Automation.PowerShell.AddStatement](#) method, which adds an additional statement to the end of the pipeline. The following code gets a list of running processes with the name `powershell`, and then gets the list of running services.

C#

```
PowerShell ps = PowerShell.Create();
ps.AddCommand("Get-Process").AddParameter("Name", "powershell");
ps.AddStatement().AddCommand("Get-Service");
ps.Invoke();
```

AddScript

You can run an existing script by calling the [System.Management.Automation.PowerShell.AddScript](#) method. The following example adds a script to the pipeline and runs it. This example assumes there is already a script named `MyScript.ps1` in a folder named `D:\PSScripts`.

C#

```
PowerShell ps = PowerShell.Create();
ps.AddScript("D:\PSScripts\MyScript.ps1").Invoke();
```

There is also a version of the `AddScript` method that takes a boolean parameter named `useLocalScope`. If this parameter is set to `true`, then the script is run in the local scope. The following code will run the script in the local scope.

C#

```
PowerShell ps = PowerShell.Create();
ps.AddScript(@"D:\PSScripts\MyScript.ps1", true).Invoke();
```

Creating a custom runspace

While the default runspace used in the previous examples loads all of the core Windows PowerShell commands, you can create a custom runspace that loads only a specified subset of all commands. You might want to do this to improve performance (loading a larger number of commands is a performance hit), or to restrict the capability of the user to perform operations. A runspace that exposes only a limited number of commands is called a constrained runspace. To create a constrained runspace, you use

the [System.Management.Automation.Runspaces.Runspace](#) and [System.Management.Automation.Runspaces.InitialSessionState](#) classes.

Creating an InitialSessionState object

To create a custom runspace, you must first create a [System.Management.Automation.Runspaces.InitialSessionState](#) object. In the following example, we use the [System.Management.Automation.Runspaces.RunspaceFactory](#) to create a runspace after creating a default InitialSessionState object.

C#

```
InitialSessionState iss = InitialSessionState.CreateDefault();
Runspace rs = RunspaceFactory.CreateRunspace(iss);
rs.Open();
PowerShell ps = PowerShell.Create();
ps.Runspace = rs;
ps.AddCommand("Get-Command");
ps.Invoke();
rs.Close();
```

Constraining the runspace

In the previous example, we created a default [System.Management.Automation.Runspaces.InitialSessionState](#) object that loads all of the built-in core Windows PowerShell. We could also have called the [System.Management.Automation.Runspaces.InitialSessionState.CreateDefault2](#) method to create an InitialSessionState object that would load only the commands in the Microsoft.PowerShell.Core snapin. To create a more constrained runspace, you must create an empty InitialSessionState object by calling the [System.Management.Automation.Runspaces.InitialSessionState.Create](#) method, and then add commands to the InitialSessionState.

Using a runspace that loads only the commands that you specify provides significantly improved performance.

You use the methods of the [System.Management.Automation.Runspaces.SessionStateCmdletEntry](#) class to define cmdlets for the initial session state. The following example creates an empty initial session state, then defines and adds the `Get-Command` and `Import-Module` commands to the initial session state. We then create a runspace constrained by that initial session state, and execute the commands in that runspace.

Create the initial session state.

```
C#
```

```
InitialSessionState iss = InitialSessionState.Create();
```

Define and add commands to the initial session state.

```
C#
```

```
SessionStateCmdletEntry getCommand = new SessionStateCmdletEntry(
    "Get-Command", typeof(Microsoft.PowerShell.Commands.GetCommandCommand),
    "");
SessionStateCmdletEntry importModule = new SessionStateCmdletEntry(
    "Import-Module",
    typeof(Microsoft.PowerShell.Commands.ImportModuleCommand), "");
iss.Commands.Add(getCommand);
iss.Commands.Add(importModule);
```

Create and open the runspace.

```
C#
```

```
Runspace rs = RunspaceFactory.CreateRunspace(iss);
rs.Open();
```

Execute a command and show the result.

```
C#
```

```
PowerShell ps = PowerShell.Create();
ps.Runspace = rs;
ps.AddCommand("Get-Command");
Collection<CommandInfo> result = ps.Invoke<CommandInfo>();
foreach (var entry in result)
{
    Console.WriteLine(entry.Name);
}
```

Close the runspace.

```
C#
```

```
rs.Close();
```

When run, the output of this code will look as follows.

PowerShell

```
Get-Command  
Import-Module
```

Creating Runspaces

Article • 03/24/2025

A runspace is the operating environment for the commands that are invoked by a host application. This environment includes the commands and data that are currently present, and any language restrictions that currently apply.

Host applications can use the default runspace that is provided by Windows PowerShell, which includes all available core commands, or create a custom runspace that includes only a subset of the available commands. To create a customized runspace, you create a [System.Management.Automation.Runspaces.InitialSessionState](#) object and assign it to your runspace.

Runspace tasks

1. [Creating an InitialSessionState](#)
2. [Creating a constrained runspace](#)
3. [Creating multiple runspaces](#)

See Also

Creating an InitialSessionState

Article • 12/18/2023

PowerShell commands run in a runspace. To host PowerShell in your application, you must create a `System.Management.Automation.Runspaces.Runspace` object. Every runspace has a `System.Management.Automation.Runspaces.InitialSessionState` object associated with it. The `InitialSessionState` specifies characteristics of the runspace, such as which commands, variables, and modules are available for that runspace.

Create a default InitialSessionState

The `CreateDefault` and `CreateDefault2` methods of the `InitialSessionState` class can be used to create an `InitialSessionState` object. The `CreateDefault` method creates an `InitialSessionState` with all of the built-in commands loaded, while the `CreateDefault2` method loads only the commands required to host PowerShell (the commands from the `Microsoft.PowerShell.Core` module).

If you want to further limit the commands available in your host application you need to create a constrained runspace. For information, see [Creating a constrained runspace](#).

The following code shows how to create an `InitialSessionState`, assign it to a runspace, add commands to the pipeline in that runspace, and invoke the commands. For more information about adding and invoking commands, see [Adding and invoking commands](#).

C#

```
namespace SampleHost
{
    using System;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;

    class HostP4b
    {
        static void Main(string[] args)
        {
            // Call InitialSessionState.CreateDefault() to create an empty
            // InitialSessionState object, then add the variables that will be
            // available when the runspace is opened.
            InitialSessionState iss = InitialSessionState.CreateDefault();
            SessionStateVariableEntry var1 =
                new SessionStateVariableEntry("test1",
                                              "MyVar1",
                                              "Initial session state MyVar1 test");
        }
    }
}
```

```
iss.Variables.Add(var1);

SessionStateVariableEntry var2 =
    new SessionStateVariableEntry("test2",
        "MyVar2",
        "Initial session state MyVar2 test");
iss.Variables.Add(var2);

// Call RunspaceFactory.CreateRunspace(InitialSessionState) to
// create the runspace where the pipeline is run.
Runspace rs = RunspaceFactory.CreateRunspace(iss);
rs.Open();

// Call PowerShell.Create() to create the PowerShell object, then
// specify the runspace and pipeline commands.
PowerShell ps = PowerShell.Create();
ps.Runspace = rs;
ps.AddCommand("Get-Variable");
ps.AddArgument("test*");

Console.WriteLine("Variable          Value");
Console.WriteLine("-----");

// Call ps.Invoke() to run the pipeline synchronously.
foreach (PSObject result in ps.Invoke())
{
    Console.WriteLine("{0,-20}{1}",
        result.Members["Name"].Value,
        result.Members["Value"].Value);
} // End foreach.

// Close the runspace to free resources.
rs.Close();

} // End Main.
} // End SampleHost.
}
```

See Also

[Creating a constrained runspace](#)

[Adding and invoking commands](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Creating a constrained runspace

Article • 03/24/2025

For performance or security reasons, you might want to restrict the Windows PowerShell commands available to your host application. To do this you create an empty [System.Management.Automation.Runspaces.InitialSessionState](#) by calling the [System.Management.Automation.Runspaces.InitialSessionState.Create*](#) method, and then add only the commands you want available.

Using a runspace that loads only the commands that you specify provides significantly improved performance.

You use the methods of the [System.Management.Automation.Runspaces.SessionStateCmdletEntry](#) class to define cmdlets for the initial session state.

You can also make commands private. Private commands can be used by the host application, but not by users of the application.

Adding commands to an empty runspace

The following example demonstrates how to create an empty InitialSessionState and add commands to it.

C#

```
namespace Microsoft.Samples.PowerShell.Runspace
{
    using System;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspace;
    using Microsoft.PowerShell.Commands;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for the application.
    /// </summary>
    internal class Runspace10b
    {
        /// <summary>
        /// This sample shows how to create an empty initial session state,
        /// how to add commands to the session state, and then how to create a
        /// runspace that has only those two commands. A PowerShell object
        /// is used to run the Get-Command cmdlet to show that only two commands
        /// are available.
    }
}
```

```
/// </summary>
/// <param name="args">Parameter not used.</param>
private static void Main(string[] args)
{
    // Create an empty InitialSessionState and then add two commands.
    InitialSessionState iss = InitialSessionState.Create();

    // Add the Get-Process and Get-Command cmdlets to the session state.
    SessionStateCmdletEntry ssce1 = new SessionStateCmdletEntry(
        "Get-Process",

        typeof(GetProcessCommand),
        null);
    iss.Commands.Add(ssce1);

    SessionStateCmdletEntry ssce2 = new SessionStateCmdletEntry(
        "Get-Command",
        typeof(GetCommandCommand),
        null);
    iss.Commands.Add(ssce2);

    // Create a runspace.
    using (Runspace myRunSpace = RunspaceFactory.CreateRunspace(iss))
    {
        myRunSpace.Open();
        using (PowerShell powershell = PowerShell.Create())
        {
            powershell.Runspace = myRunSpace;

            // Create a pipeline with the Get-Command command.
            powershell.AddCommand("Get-Command");

            Collection<PSObject> results = powershell.Invoke();

            Console.WriteLine("Verb           Noun");
            Console.WriteLine("-----");

            // Display each result object.
            foreach (PSObject result in results)
            {
                Console.WriteLine(
                    "{0,-20} {1}",
                    result.Members["verb"].Value,
                    result.Members["Noun"].Value);
            }
        }

        // Close the runspace and release any resources.
        myRunSpace.Close();
    }

    System.Console.WriteLine("Hit any key to exit...");
    System.Console.ReadKey();
}
```

```
}
```

Making commands private

You can also make a command private, by setting it's [System.Management.Automation.CommandInfo.Visibility](#) property to [System.Management.Automation.SessionStateEntryVisibility](#) **Private**. The host application and other commands can call that command, but the user of the application cannot. In the following example, the [Get-ChildItem](#) command is private.

C#

```
defaultSessionState = InitialSessionState.CreateDefault();
commandIndex = GetIndexOfEntry(defaultSessionState.Commands, "Get-
ChildItem");
defaultSessionState.Commands[commandIndex].Visibility =
SessionStateEntryVisibility.Private;

this.runspace = RunspaceFactory.CreateRunspace(defaultSessionState);
this.runspace.Open();
```

See Also

[Creating an InitialSessionState](#)

Creating multiple runspaces

Article • 09/17/2021

If you create a large number of runspaces, you might consider creating a runspace pool. Using a [System.Management.Automation.Runspaces.RunspacePool](#) object, rather than creating a large number of individual runspaces with the same characteristics, can improve performance.

Creating and using a runspace pool.

The following example shows how to create a runspace pool and how to run a command asynchronously in a runspace of the pool.

C#

```
namespace HostRunspacePool
{
    using System;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;

    /// <summary>
    /// This class provides the Main entry point for the Host application.
    /// </summary>
    internal class HostRunspacePool
    {
        /// <summary>
        /// This sample demonstrates the following.
        /// 1. Creating and opening a runspace pool.
        /// 2. Creating a PowerShell object.
        /// 3. Adding commands and arguments to the PowerShell object.
        /// 4. Running the commands asynchronously using the runspace
        ///     of the runspace pool.
        /// </summary>
        /// <param name="args">Parameter is not used.</param>
        private static void Main(string[] args)
        {
            // Create a pool of runspaces.
            using (RunspacePool rsp = RunspaceFactory.CreateRunspacePool())
            {
                rsp.Open();

                // Create a PowerShell object to run the following command.
                // Get-Process wmi*
                PowerShell gpc = PowerShell.Create();
                // Specify the runspace to use and add commands.
                gpc.RunspacePool = rsp;
```

```
gpc.AddCommand("Get-Process").AddArgument("wmi*");

    // Invoke the command asynchronously.
    IAsyncResult gpcAsyncResult = gpc.BeginInvoke();
    // Get the results of running the command.
    PSDataCollection<PSObject> gpcOutput =
gpc.EndInvoke(gpcAsyncResult);

    // Process the output.
    Console.WriteLine("The output from running the command: Get-Process
wmi*");
    for (int i= 0; i < gpcOutput.Count; i++)
{
    Console.WriteLine(
        "Process Name: {0} Process Id: {1}",
        gpcOutput[i].Properties["ProcessName"].Value,
        gpcOutput[i].Properties["Id"].Value);
}
} // End using.
} // End Main entry point.
} // End HostPs5 class.
}
```

See Also

[Creating an InitialSessionState](#)

Adding and invoking commands

Article • 03/24/2025

After creating a runspace, you can add Windows PowerShell commands and scripts to a pipeline, and then invoke the pipeline synchronously or asynchronously.

Creating a pipeline

The [System.Management.Automation.PowerShell](#) class provides several methods to add commands, parameters, and scripts to the pipeline. You can invoke the pipeline synchronously by calling an overload of the [System.Management.Automation.PowerShell.Invoke*](#) method, or asynchronously by calling an overload of the [System.Management.Automation.PowerShell.BeginInvoke*](#) and then the [System.Management.Automation.PowerShell.EndInvoke*](#) method.

AddCommand

1. Create a [System.Management.Automation.PowerShell](#) object.

```
C#  
  
PowerShell ps = PowerShell.Create();
```

2. Add the command that you want to execute.

```
C#  
  
ps.AddCommand("Get-Process");
```

3. Invoke the command.

```
C#  
  
ps.Invoke();
```

If you call the [System.Management.Automation.PowerShell.AddCommand*](#) method more than once before you call the [System.Management.Automation.PowerShell.Invoke*](#) method, the result of the first command is piped to the second, and so on. If you do not want to pipe the result of a

previous command to a command, add it by calling the [System.Management.Automation.PowerShell.AddStatement*](#) instead.

AddParameter

The previous example executes a single command without any parameters. You can add parameters to the command by using the [System.Management.Automation.PSCmdlet.AddParameter*](#) method. For example, the following code gets a list of all of the processes that are named `powershell` running on the machine.

C#

```
PowerShell.Create().AddCommand("Get-Process")
    .AddParameter("Name", "powershell")
    .Invoke();
```

You can add additional parameters by calling

[System.Management.Automation.PSCmdlet.AddParameter*](#) repeatedly.

C#

```
PowerShell.Create().AddCommand("Get-Command")
    .AddParameter("Name", "Get-VM")
    .AddParameter("Module", "Hyper-V")
    .Invoke();
```

You can also add a dictionary of parameter names and values by calling the

[System.Management.Automation.PowerShell.AddParameters*](#) method.

C#

```
IDictionary parameters = new Dictionary<String, String>();
parameters.Add("Name", "Get-VM");

parameters.Add("Module", "Hyper-V");
PowerShell.Create().AddCommand("Get-Command")
    .AddParameters(parameters)
    .Invoke()
```

AddStatement

You can simulate batching by using the [System.Management.Automation.PowerShell.AddStatement](#)* method, which adds an additional statement to the end of the pipeline. The following code gets a list of running processes with the name `powershell`, and then gets the list of running services.

C#

```
PowerShell ps = PowerShell.Create();
ps.AddCommand("Get-Process").AddParameter("Name", "powershell");
ps.AddStatement().AddCommand("Get-Service");
ps.Invoke();
```

AddScript

You can run an existing script by calling the [System.Management.Automation.PowerShell.AddScript](#)* method. The following example adds a script to the pipeline and runs it. This example assumes there is already a script named `MyScript.ps1` in a folder named `D:\PSScripts`.

C#

```
PowerShell ps = PowerShell.Create();
ps.AddScript(File.ReadAllText(@"D:\PSScripts\MyScript.ps1")).Invoke();
```

There is also a version of the [System.Management.Automation.PowerShell.AddScript](#)* method that takes a boolean parameter named `useLocalScope`. If this parameter is set to `true`, then the script is run in the local scope. The following code will run the script in the local scope.

C#

```
PowerShell ps = PowerShell.Create();
ps.AddScript(File.ReadAllText(@"D:\PSScripts\MyScript.ps1"), true).Invoke();
```

Invoking a pipeline synchronously

After you add elements to the pipeline, you invoke it. To invoke the pipeline synchronously, you call an overload of the [System.Management.Automation.PowerShell.Invoke](#)* method. The following example shows how to synchronously invoke a pipeline.

C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Management.Automation;

namespace HostPS1e
{
    class HostPS1e
    {
        static void Main(string[] args)
        {
            // Using the PowerShell.Create and AddCommand
            // methods, create a command pipeline.
            PowerShell ps = PowerShell.Create().AddCommand ("Sort-Object");

            // Using the PowerShell.Invoke method, run the command
            // pipeline using the supplied input.
            foreach (PSObject result in ps.Invoke(new int[] { 3, 1, 6, 2, 5, 4 }))
            {
                Console.WriteLine("{0}", result);
            } // End foreach.
        } // End Main.
    } // End HostPS1e.
}

```

Invoking a pipeline asynchronously

You invoke a pipeline asynchronously by calling an overload of the [System.Management.Automation.PowerShell.BeginInvoke*](#) to create an [IAsyncResult](#) object, and then calling the [System.Management.Automation.PowerShell.EndInvoke*](#) method.

The following example shows how to invoke a pipeline asynchronously.

C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Management.Automation;

namespace HostPS3
{
    class HostPS3
    {
        static void Main(string[] args)
        {
            // Use the PowerShell.Create and PowerShell.AddCommand

```

```
// methods to create a command pipeline that includes
// Get-Process cmdlet. Do not include spaces immediately
// before or after the cmdlet name as that will cause
// the command to fail.
PowerShell ps = PowerShell.Create().AddCommand("Get-Process");

// Create an IAsyncResult object and call the
// BeginInvoke method to start running the
// command pipeline asynchronously.
IAsyncResult asyncpl = ps.BeginInvoke();

// Using the PowerShell.Invoke method, run the command
// pipeline using the default runspace.
foreach (PSObject result in ps.EndInvoke(asyncpl))
{
    Console.WriteLine("{0,-20}{1}",
        result.Members["ProcessName"].Value,
        result.Members["Id"].Value);
} // End foreach.
System.Console.WriteLine("Hit any key to exit.");
System.Console.ReadKey();
} // End Main.
} // End HostPS3.
}
```

See Also

[Creating an InitialSessionState](#)

[Creating a constrained runspace](#)

Creating remote runspaces

Article • 09/17/2021

PowerShell commands that take a `ComputerName` parameter can be run on any computer that runs PowerShell. To run commands that don't take a `ComputerName` parameter, you can use WS-Management to configure a runspace that connects to a specified computer, and run commands on that computer.

Using a `WSManConnection` to create a remote runspace

To create a runspace that connects to a remote computer, you create a `System.Management.Automation.Runspaces.WSManConnectionInfo` object. You specify the target endpoint for the connection by setting the `System.Management.Automation.Runspaces.WSManConnectionInfo.ConnectionUri` property of the object. You then create a runspace by calling the `System.Management.Automation.Runspaces.RunspaceFactory.CreateRunspace` method, specifying the `System.Management.Automation.Runspaces.WSManConnectionInfo` object as the `connectionInfo` parameter.

The following example shows how to create a runspace that connects to a remote computer. In the example, `RemoteComputerUri` is used as a placeholder for the actual URI of a remote computer.

C#

```
namespace Samples
{
    using System;
    using System.Collections.ObjectModel;
    using System.Management.Automation; // PowerShell namespace.
    using System.Management.Automation.Runspaces; // PowerShell namespace.

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class RemoteRunspace02
    {
        /// <summary>
        /// This sample shows how to create a remote runspace that
        /// runs commands on the local computer.
        /// </summary>
        /// <param name="args">Parameter not used.</param>
        private static void Main(string[] args)
```

```

{
    // Create a WSMANConnectionInfo object using the default constructor
    // to connect to the "localHost". The WSMANConnectionInfo object can
    // also be used to specify connections to remote computers.
    Uri RemoteComputerUri = new Uri("http://Server01:5985/WSMAN");
    WSMANConnectionInfo connectionInfo = new
    WSMANConnectionInfo(RemoteComputerUri);

        // Set the OperationTimeout property and OpenTimeout properties.
        // The OperationTimeout property is used to tell PowerShell
        // how long to wait (in milliseconds) before timing out for an
        // operation. The OpenTimeout property is used to tell Windows
        // PowerShell how long to wait (in milliseconds) before timing out
        // while establishing a remote connection.
    connectionInfo.OperationTimeout = 4 * 60 * 1000; // 4 minutes.
    connectionInfo.OpenTimeout = 1 * 60 * 1000; // 1 minute.

        // Create a remote runspace using the connection information.
        //using (Runspace remoteRunspace = RunspaceFactory.CreateRunspace())
        using (Runspace remoteRunspace =
    RunspaceFactory.CreateRunspace(connectionInfo))
    {
        // Establish the connection by calling the Open() method to open the
        runspace.
        // The OpenTimeout value set previously will be applied while
        establishing
        // the connection. Establishing a remote connection involves sending
        and
        // receiving some data, so the OperationTimeout will also play a
        role in this process.
        remoteRunspace.Open();

        // Create a PowerShell object to run commands in the remote
        runspace.
        using (PowerShell powershell = PowerShell.Create())
        {
            powershell.Runspace = remoteRunspace;
            powershell.AddCommand("Get-Process");
            powershell.Invoke();

            Collection<PSObject> results = powershell.Invoke();

            Console.WriteLine("Process                  HandleCount");
            Console.WriteLine("-----");

            // Display the results.
            foreach (PSObject result in results)
            {
                Console.WriteLine(
                    "{0,-20} {1}",
                    result.Members["ProcessName"].Value,
                    result.Members["HandleCount"].Value);
            }
        }
    }
}

```

```
// Close the connection. Call the Close() method to close the remote
// runspace. The Dispose() method (called by using primitive) will
call
    // the Close() method if it is not already called.
    remoteRunspace.Close();
}
}
}
}
```

Creating a custom user interface

Article • 03/24/2025

Windows PowerShell provides abstract classes and interfaces that allow you to create a custom interactive UI that hosts the Windows PowerShell engine. To create a custom UI, you must implement the [System.Management.Automation.Host.PSHost](#) class.

Optionally, you can also implement the

[System.Management.Automation.Host.PSHostRawUserInterface](#) and

[System.Management.Automation.Host.PSHostUserInterface](#) classes, and the

[System.Management.Automation.Host.IHostSupportsInteractiveSession](#) and

[System.Management.Automation.Host.IHostUISupportsMultipleChoiceSelection](#)

interfaces.

Host Application Samples

Article • 09/17/2021

This section includes sample code that is provided in the Windows PowerShell 2.0 SDK.

In This Section

[PowerShell API Samples](#) This section includes sample code that shows how to create runspaces that restrict functionality, and how to asynchronously run commands using a runspace pool to supply the runspaces.

[Custom Host Samples](#) Includes sample code for writing a custom host. The host is the component of Windows PowerShell that provides communications between the user and the Windows PowerShell engine. For more information about custom hosts, see [Custom Host](#).

[Runspace Samples](#) Includes sample code for creating runspaces. For more information about how runspaces are used, see [Host Application Runspaces](#).

[Remote Runspace Samples](#) This section includes sample code that shows how to create runspaces that can be used to connect to a computer by using WS-Management-based Windows PowerShell remoting.

See Also

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Windows PowerShell API Samples

Article • 12/18/2023

This section includes sample code that shows how to create runspaces that restrict functionality, and how to asynchronously run commands by using a runspace pool to supply the runspaces. You can use Microsoft Visual Studio to create a console application and then copy the code from the topics in this section into your host application.

In This Section

[PowerShell01 Sample](#) This sample shows how to use a `System.Management.Automation.Runspaces.InitialSessionState` object to limit the functionality of a runspace. The output of this sample demonstrates how to restrict the language mode of the runspace, how to mark a cmdlet as private, how to add and remove cmdlets and providers, how to add a proxy command, and more.

[PowerShell02 Sample](#) This sample shows how to run commands asynchronously by using the runspaces of a runspace pool. The sample generates a list of commands, and then runs those commands while the Windows PowerShell engine opens a runspace from the pool when it is needed.

Windows PowerShell01 Sample

Article • 03/24/2025

This sample shows how to use a [System.Management.Automation.Runspaces.InitialSessionState](#) object to limit the functionality of a runspace. The output of this sample demonstrates how to restrict the language mode of the runspace, how to mark a cmdlet as private, how to add and remove cmdlets and providers, how to add a proxy command, and more. This sample concentrates on how to restrict the runspace programmatically. Scripting alternatives to restricting the runspace include the `$ExecutionContext.SessionState.LanguageMode` and `PSSessionConfiguration` commands.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following:

- Restricting the language by setting the [System.Management.Automation.Runspaces.InitialSessionState.LanguageMode](#) property.
- Adding aliases to the initial session state by using a [System.Management.Automation.Runspaces.SessionStateAliasEntry](#) object.
- Marking commands as private.
- Removing providers from the initial session state by using the [System.Management.Automation.Runspaces.InitialSessionState.Providers](#) property.
- Removing commands from the initial session state by using the [System.Management.Automation.Runspaces.InitialSessionState.Commands](#) property.
- Adding commands and providers to the [System.Management.Automation.Runspaces.InitialSessionState](#) object.

Example

This sample shows several ways to limit the functionality of a runspace.

C#

```
namespace Sample
{
    using System;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;

    /// <summary>
    /// This class contains the Main entry point for the application.
    /// </summary>
    internal class PowerShell01
    {
        /// <summary>
        /// The runspace used to run commands.
        /// </summary>
        private Runspace runspace;

        /// <summary>
        /// Return the first index of the entry in <paramref name="entries"/>
        /// with the name <paramref name="name"/>. Return -1 if it is not found.
        /// </summary>
        /// <typeparam name="T">Type of ConstrainedSessionStateEntry</typeparam>
        /// <param name="entries">Collection of entries to search for <paramref
        name="name"/> in.</param>
        /// <param name="name">Named of the entry we are looking for</param>
        /// <returns>
        /// The first index of the entry in <paramref name="entries"/> with the
        /// name <paramref name="name"/>, or return -1 if it is not found.
        /// </returns>
        private static int GetIndexOfEntry<T>(
            InitialSessionStateEntryCollection<T> entries,
            string name) where T : ConstrainedSessionStateEntry
        {
            int foundIndex = 0;
            foreach (T entry in entries)
            {
                if (entry.Name.Equals(name, StringComparison.OrdinalIgnoreCase))
                {
                    return foundIndex;
                }

                foundIndex++;
            }

            return -1;
        }

        /// <summary>
        /// Run commands to demonstrate the ways to constrain the runspace.
        /// </summary>
    }
}
```

```
/// <param name="args">This parameter is unused.</param>
private static void Main(string[] args)
{
    new PowerShell01().RunCommands();
}

/// <summary>
/// Run a script to display the results and errors.
/// </summary>
/// <param name="script">Script to be run.</param>
/// <param name="scriptComment">Comment to be printed about
/// the script.</param>
private void RunScript(string script, string scriptComment)
{
    Console.WriteLine("Running '{0}'\n{1}.\n\nPowerShell Output:", script,
scriptComment);

    // Using a PowerShell object, create a pipeline, add the script to the
    // pipeline, and specify the runspace where the pipeline is invoked.
    PowerShell powerShellCommand = PowerShell.Create();
    powerShellCommand.AddScript(script);
    powerShellCommand.Runspace = this.runspace;

    try
    {
        Collection<PSObject> results = powerShellCommand.Invoke();

        // Display the results.
        foreach (PSObject result in results)
        {
            Console.WriteLine(result);
        }

        // Display any non-terminating errors.
        foreach (ErrorRecord error in powerShellCommand.Streams.Error)
        {
            Console.WriteLine("PowerShell Error: {0}", error);
        }
    }
    catch (RuntimeException ex)
    {
        Console.WriteLine("PowerShell Error: {0}", ex.Message);
        Console.WriteLine();
    }

    Console.WriteLine("\n-----\n");
}

/// <summary>
/// Run some commands to demonstrate the script capabilities.
/// </summary>
private void RunCommands()
{
    this.runspace =
RunspaceFactory.CreateRunspace(InitialSessionState.CreateDefault());
```

```

        this.runspace.Open();
        this.RunScript("$a=0;$a", "Assigning to a variable will work for a
default InitialSessionState");
        this.runspace.Close();

        this.runspace =
RunspaceFactory.CreateRunspace(InitialSessionState.CreateDefault());
        this.runspace.InitialSessionState.LanguageMode =
PSLanguageMode.RestrictedLanguage;
        this.runspace.Open();
        this.RunScript("$a=0;$a", "Assigning to a variable will not work in
RestrictedLanguage LanguageMode");
        this.runspace.Close();

        this.runspace =
RunspaceFactory.CreateRunspace(InitialSessionState.CreateDefault());
        this.runspace.InitialSessionState.LanguageMode =
PSLanguageMode.NoLanguage;
        this.runspace.Open();
        this.RunScript("10/2", "A script will not work in NoLanguage
LanguageMode.");
        this.runspace.Close();

        this.runspace =
RunspaceFactory.CreateRunspace(InitialSessionState.CreateDefault());
        this.runspace.Open();
        string scriptComment = "Get-ChildItem with a default
InitialSessionState will work since the standard \n" +
            "PowerShell cmdlets are included in the default
InitialSessionState";
        this.RunScript("Get-ChildItem", scriptComment);
        this.runspace.Close();

        InitialSessionState defaultSessionState =
InitialSessionState.CreateDefault();
        defaultSessionState.Commands.Add(new SessionStateAliasEntry("dir2",
"Get-ChildItem"));
        this.runspace = RunspaceFactory.CreateRunspace(defaultSessionState);
        this.runspace.Open();
        this.RunScript("dir2", "An alias, like dir2, can be added to
InitialSessionState");
        this.runspace.Close();

        defaultSessionState = InitialSessionState.CreateDefault();
        int commandIndex = GetIndexOfEntry(defaultSessionState.Commands, "Get-
ChildItem");
        defaultSessionState.Commands.RemoveItem(commandIndex);
        this.runspace = RunspaceFactory.CreateRunspace(defaultSessionState);
        this.runspace.Open();
        scriptComment = "Get-ChildItem was removed from the list of commands
so it\nwill no longer be found";
        this.RunScript("Get-ChildItem", scriptComment);
        this.runspace.Close();

        defaultSessionState = InitialSessionState.CreateDefault();

```

```

defaultSessionState.Providers.Clear();
this.runspace = RunspaceFactory.CreateRunspace(defaultSessionState);
this.runspace.Open();
this.RunScript("Get-ChildItem", "There are no providers so Get-
ChildItem will not work");
this.runspace.Close();

// Marks a command as private, and then defines a proxy command
// that uses the private command. One reason to define a proxy for
// a command is to remove a parameter of the original command.
// For a more complete sample of a proxy command, see the Runspace11
// sample.
defaultSessionState = InitialSessionState.CreateDefault();
commandIndex = GetIndexOfEntry(defaultSessionState.Commands, "Get-
ChildItem");
defaultSessionState.Commands[commandIndex].Visibility =
SessionStateEntryVisibility.Private;
CommandMetadata getChildItemMetadata = new CommandMetadata(
    typeof(Microsoft.PowerShell.Commands.GetChildItemCommand));
getChildItemMetadata.Parameters.Remove("Recurse");
string getChildItemBody = ProxyCommand.Create(getChildItemMetadata);
defaultSessionState.Commands.Add(new SessionStateFunctionEntry("Get-
ChildItem2", getChildItemBody));
this.runspace = RunspaceFactory.CreateRunspace(defaultSessionState);
this.runspace.Open();
this.RunScript("Get-ChildItem", "Get-ChildItem is private so it will
not be available");
scriptComment = "Get-ChildItem2 is a proxy to Get-ChildItem. \n" +
    "It works even when Get-ChildItem is private.";
this.RunScript("Get-ChildItem2", scriptComment);
scriptComment = "This will fail. Unlike Get-ChildItem, Get-ChildItem2
does not have -Recurse";
this.RunScript("Get-ChildItem2 -Recurse", scriptComment);

InitialSessionState cleanSessionState = InitialSessionState.Create();
this.runspace = RunspaceFactory.CreateRunspace(cleanSessionState);
this.runspace.Open();
scriptComment = "A script will not work because \n" +
    "InitialSessionState.Create() will have the default
LanguageMode of NoLanguage";
this.RunScript("10/2", scriptComment);
this.runspace.Close();

cleanSessionState = InitialSessionState.Create();
cleanSessionState.LanguageMode = PSLanguageMode.FullLanguage;
this.runspace = RunspaceFactory.CreateRunspace(cleanSessionState);
this.runspace.Open();
scriptComment = "Get-ChildItem, standard cmdlets and providers are not
present \n" +
    "in an InitialSessionState returned from
InitialSessionState.Create()";
this.RunScript("Get-ChildItem", scriptComment);
this.runspace.Close();

cleanSessionState = InitialSessionState.Create();

```

```
cleanSessionState.Commands.Add(
    new SessionStateCmdletEntry(
        "Get-ChildItem",

typeof(Microsoft.PowerShell.Commands.GetChildItemCommand),
    null));
cleanSessionState.Providers.Add(
    new SessionStateProviderEntry(
        "FileSystem",

typeof(Microsoft.PowerShell.Commands.FileSystemProvider),
    null));
cleanSessionState.LanguageMode = PSLanguageMode.FullLanguage;
this.runspace = RunspaceFactory.CreateRunspace(cleanSessionState);
this.runspace.Open();
scriptComment = "Get-ChildItem and the FileSystem provider were
explicitly added\n" +
    "so Get-ChildItem will work";
this.RunScript("Get-ChildItem", scriptComment);
this.runspace.Close();

Console.WriteLine("Done...");
Console.ReadLine();
}
}
}
```

See Also

[Writing a Windows PowerShell Host Application](#)

Windows PowerShell02 Sample

Article • 09/17/2021

This sample shows how to run commands asynchronously using the runspaces of a runspace pool. The sample generates a list of commands, and then runs those commands while the Windows PowerShell engine opens a runspace from the pool when it is needed.

Requirements

- This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following:

- Creating a RunspacePool object with a minimum and maximum number of runspaces allowed to be open at the same time.
- Creating a list of commands.
- Running the commands asynchronously.
- Calling the [System.Management.Automation.Runspaces.RunspacePool.GetAvailableRunspaces](#) * method to see how many runspaces are free.
- Capturing the command output with the [System.Management.Automation.PowerShell.EndInvoke](#)* method.

Example

This sample shows how to open the runspaces of a runspace pool, and how to asynchronously run commands in those runspaces.

C#

```
namespace Sample
{
    using System;
    using System.Collections;
    using System.Collections.Generic;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;

    /// <summary>
```

```

/// This class contains the Main entry point for the application.
/// </summary>
internal class PowerShell02
{
    /// <summary>
    /// Runs many commands with the help of a RunspacePool.
    /// </summary>
    /// <param name="args">This parameter is unused.</param>
    private static void Main(string[] args)
    {
        // Creating and opening runspace pool. Use a minimum of 1 runspace and
        // a maximum of
        // 5 runspaces can be opened at the same time.
        RunspacePool runspacePool = RunspaceFactory.CreateRunspacePool(1, 5);
        runspacePool.Open();

        using (runspacePool)
        {
            // Define the commands to be run.
            List<PowerShell> powerShellCommands = new List<PowerShell>();

            // The command results.
            List<IAsyncResult> powerShellCommandResults = new List<IAsyncResult>();
        }

        // The maximum number of runspaces that can be opened at one time is
        // 5, but we can queue up many more commands that will use the
        // runspace pool.
        for (int i = 0; i < 100; i++)
        {
            // Using a PowerShell object, run the commands.
            PowerShell powershell = PowerShell.Create();

            // Instead of setting the Runspace property of powershell,
            // the RunspacePool property is used. That is the only difference
            // between running commands with a runspace and running commands
            // with a runspace pool.
            powershell.RunspacePool = runspacePool;

            // The script to be run outputs a sequence number and the number
            // of available runspaces
            // in the pool.
            string script = String.Format(
                "write-output ' Command: {0}, Available Runspaces:
{1}'",
                i,
                runspacePool.GetAvailableRunspaces());

            // The three lines below look the same running with a runspace or
            // with a runspace pool.
            powershell.AddScript(script);
            powerShellCommands.Add(powershell);
            powerShellCommandResults.Add(powershell.BeginInvoke());
        }
    }
}

```

```

    // Collect the results.
    for (int i = 0; i < 100; i++)
    {
        // EndInvoke will wait for each command to finish, so we will be
        getting the commands
        // in the same 0 to 99 order that they have been invoked with
        BeginInvoke.

        PSDataCollection<PSObject> results =
powerShellCommands[i].EndInvoke(powerShellCommandResults[i]);

        // Print all the results. One PSObject with a plain string is the
        expected result.
        PowerShell02.PrintCollection(results);
    }
}

/// <summary>
/// Iterates through a collection printing all items.
/// </summary>
/// <param name="collection">collection to be printed</param>
private static void PrintCollection(IList collection)
{
    foreach (object obj in collection)
    {
        Console.WriteLine("PowerShell Result: {0}", obj);
    }
}
}

```

See Also

[Writing a Windows PowerShell Host Application](#)

Custom Host Samples

Article • 09/17/2021

This section includes sample code for writing a custom host. You can use Microsoft Visual Studio to create a console application and then copy the code from the topics in this section into your host application.

In This Section

[Host01 Sample](#) This sample shows how to implement a host application that uses a basic custom host.

[Host02 Sample](#) This sample shows how to write a host application that uses the Windows PowerShell runtime along with a custom host implementation. The host application sets the host culture to German, runs the [Get-Process](#) cmdlet and displays the results as you would see them using pwrsh.exe, and then prints out the current date and time in German.

[Host03 Sample](#) This sample shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console.

[Host04 Sample](#) This sample shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console. This host application also supports displaying prompts that allow the user to specify multiple choices.

[Host05 Sample](#) This sample shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console. This host application also supports calls to remote computers by using the [Enter-PSSession](#) and [Exit-PSSession](#) cmdlets

[Host06 Sample](#) This sample shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console. In addition, this sample uses the Tokenizer APIs to specify the color of the text that is entered by the user.

See Also

Host01 Sample

Article • 04/10/2024

This sample shows how to implement a host application that uses a custom host. In this sample a runspace is created that uses the custom host, and then the [System.Management.Automation.PowerShell](#) API is used to run a script that calls "exit." The host application then looks at the output of the script and prints out the results.

This sample uses the default UI features provided by Windows PowerShell. For more information about implementing the UI features of a custom host, see [Host02 Sample](#).

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

- Creating a custom host class that derives from the [System.Management.Automation.Host.PSHost](#) class.
- Creating a runspace that uses the custom host class.
- Creating a [System.Management.Automation.PowerShell](#) object that runs a script that calls exit.
- Verifying that the correct exit code was used in the exit process.

Example 1

The following code shows an implementation of a host application that uses a simple custom host interface.

C#

```
namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
```

```
/// This class contains the Main entry point for this host application.  
/// </summary>  
internal class Host01  
{  
  
    /// <summary>  
    /// Indicator to tell the host application that it should exit.  
    /// </summary>  
    private bool shouldExit;  
  
    /// <summary>  
    /// The exit code that the host application will use to exit.  
    /// </summary>  
    private int exitCode;  
  
    /// <summary>  
    /// Gets or sets a value indicating whether the  
    /// host application should exit.  
    /// </summary>  
    public bool ShouldExit  
    {  
        get { return this.shouldExit; }  
        set { this.shouldExit = value; }  
    }  
  
    /// <summary>  
    /// Gets or sets the PSHost implementation that is  
    /// used to tell the host application what code to use  
    /// when exiting.  
    /// </summary>  
    public int ExitCode  
    {  
        get { return this.exitCode; }  
        set { this.exitCode = value; }  
    }  
  
    /// <summary>  
    /// This sample uses a PowerShell object to run  
    /// a script that calls exit. The host application looks at  
    /// this and prints out the result.  
    /// </summary>  
    /// <param name="args">Parameter not used.</param>  
    private static void Main(string[] args)  
    {  
        // Create an instance of this host application class so that  
        // the Windows PowerShell engine will have access to the  
        // ShouldExit and ExitCode parameters.  
        Host01 me = new Host01();  
  
        // Create the host instance to use.  
        MyHost myHost = new MyHost(me);  
  
        // Create a runspace that uses the host object and run the  
        // script using a PowerShell object.  
        using (Runspace myRunSpace =
```

```

RunspaceFactory.CreateRunspace(myHost))
{
    // Open the runspace.
    myRunSpace.Open();

    // Create a PowerShell object to run the script.
    using (PowerShell powershell = PowerShell.Create())
    {
        powershell.Runspace = myRunSpace;

        // Create the pipeline and run the script
        // "exit (2+2)".
        string script = "exit (2+2)";
        powershell.AddScript(script);
        powershell.Invoke(script);
    }

    // Check the flags and see if they were set properly.
    Console.WriteLine(
        "ShouldExit={0} (should be True); ExitCode={1} (should
be 4)",
        me.ShouldExit,
        me.ExitCode);

    // close the runspace to free resources.
    myRunSpace.Close();
}

Console.WriteLine("Hit any key to exit...");
Console.ReadKey();
}
}

```

Example 2

The following code is the implementation of the [System.Management.Automation.Host.PSHost](#) class that is used by this host application. Those elements that are not implemented throw an exception or return nothing.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Globalization;
    using System.Management.Automation.Host;

    /// <summary>
    /// This is a sample implementation of the PSHost abstract class for
    /// console applications. Not all members are implemented. Those that

```

```
/// are not implemented throw a NotImplementedException exception or
/// return nothing.
/// </summary>
internal class MyHost : PSHost
{
    /// <summary>
    /// A reference to the PSHost implementation.
    /// </summary>
    private Host01 program;

    /// <summary>
    /// The culture information of the thread that created
    /// this object.
    /// </summary>
    private CultureInfo originalCultureInfo =
        System.Threading.Thread.CurrentThread.CurrentCulture;

    /// <summary>
    /// The UI culture information of the thread that created
    /// this object.
    /// </summary>
    private CultureInfo originalUICultureInfo =
        System.Threading.Thread.CurrentThread.CurrentCulture;

    /// <summary>
    /// The identifier of this PSHost implementation.
    /// </summary>
    private Guid myId = Guid.NewGuid();

    /// <summary>
    /// Initializes a new instance of the MyHost class. Keep
    /// a reference to the host application object so that it
    /// can be informed of when to exit.
    /// </summary>
    /// <param name="program">
    /// A reference to the host application object.
    /// </param>
    public MyHost(Host01 program)
    {
        this.program = program;
    }

    /// <summary>
    /// Return the culture information to use. This implementation
    /// returns a snapshot of the culture information of the thread
    /// that created this object.
    /// </summary>
    public override System.Globalization.CultureInfo CurrentCulture
    {
        get { return this.originalCultureInfo; }
    }

    /// <summary>
    /// Return the UI culture information to use. This implementation
    /// returns a snapshot of the UI culture information of the thread
```

```
    /// that created this object.
    /// </summary>
    public override System.Globalization.CultureInfo CurrentUICulture
    {
        get { return this.originalUICultureInfo; }
    }

    /// <summary>
    /// This implementation always returns the GUID allocated at
    /// instantiation time.
    /// </summary>
    public override Guid InstanceId
    {
        get { return this.myId; }
    }

    /// <summary>
    /// Return a string that contains the name of the host
implementation.
    /// Keep in mind that this string may be used by script writers to
    /// identify when your host is being used.
    /// </summary>
    public override string Name
    {
        get { return "MySampleConsoleHostImplementation"; }
    }

    /// <summary>
    /// This sample does not implement a PSHostUserInterface component
so
    /// this property simply returns null.
    /// </summary>
    public override PSHostUserInterface UI
    {
        get { return null; }
    }

    /// <summary>
    /// Return the version object for this application. Typically this
    /// should match the version resource in the application.
    /// </summary>
    public override Version Version
    {
        get { return new Version(1, 0, 0, 0); }
    }

    /// <summary>
    /// Not implemented by this example class. The call fails with
    /// a NotImplementedException exception.
    /// </summary>
    public override void EnterNestedPrompt()
    {
        throw new NotImplementedException(
            "The method or operation is not implemented.");
    }
```

```
/// <summary>
/// Not implemented by this example class. The call fails
/// with a NotImplementedException exception.
/// </summary>
public override void ExitNestedPrompt()
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API is called before an external application process is
/// started. Typically it is used to save state so the parent can
/// restore state that has been modified by a child process (after
/// the child exits). In this example, this functionality is not
/// needed so the method returns nothing.
/// </summary>
public override void NotifyBeginApplication()
{
    return;
}

/// <summary>
/// This API is called after an external application process
finishes.
/// Typically it is used to restore state that a child process may
/// have altered. In this example, this functionality is not
/// needed so the method returns nothing.
/// </summary>
public override void NotifyEndApplication()
{
    return;
}

/// <summary>
/// Indicate to the host application that exit has
/// been requested. Pass the exit code that the host
/// application should use when exiting the process.
/// </summary>
/// <param name="exitCode">The exit code to use.</param>
public override void SetShouldExit(int exitCode)
{
    this.program.ShouldExit = true;
    this.program.ExitCode = exitCode;
}
}
```

See Also

Host02 Sample

Article • 03/24/2025

This sample shows how to write a host application that uses the Windows PowerShell runtime along with a custom host implementation. The host application sets the host culture to German, runs the [Get-Process](#) cmdlet and displays the results as you would see them by using pwrsh.exe, and then prints out the current date and time in German.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

- Creating a custom host whose classes derive from the [System.Management.Automation.Host.PSHost](#) class, the [System.Management.Automation.Host.PSHostUserInterface](#) class, and the [System.Management.Automation.Host.PSHostRawUserInterface](#) class.
- Creating a runspace that uses the custom host.
- Setting the host culture to German.
- Creating a [System.Management.Automation.PowerShell](#) object that runs a script to retrieve and sort the processes, then retrieves the current date which is displayed in German.

Example 1

The following code shows an implementation of a host application that uses the custom host.

C#

```
// Copyright (c) 2006 Microsoft Corporation. All rights reserved.  
//  
// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF  
// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO  
// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A  
// PARTICULAR PURPOSE.  
  
using System;
```

```
using System.Collections.Generic;
using System.Text;
using System.Management.Automation;
using System.Management.Automation.Host;
using System.Management.Automation.Runspaces;
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Host
{
    class Host02
    {
        /// <summary>
        /// Define the property that the PSHost implementation will
        /// use to tell the host application that it should exit.
        /// </summary>
        public bool ShouldExit
        {
            get { return shouldExit; }
            set { shouldExit = value; }
        }
        private bool shouldExit;

        /// <summary>
        /// Define the property that the PSHost implementation will
        /// use to tell the host application what exit code to use
        /// when exiting.
        /// </summary>
        public int ExitCode
        {
            get { return exitCode; }
            set { exitCode = value; }
        }
        private int exitCode;

        /// <summary>
        /// This sample uses the PowerShell runtime along with a host
        /// implementation to call Get-Process and display the results
        /// as you would see them in powershell.exe.
        /// </summary>
        /// <param name="args">Ignored</param>
        static void Main(string[] args)
        {
            // Set the current culture to German. We want this to be picked up
            when the MyHost
            // instance is created...
            System.Threading.Thread.CurrentThread.CurrentCulture =
            CultureInfo.GetCultureInfo("de-de");

            // Create the runspace so that you can access the pipeline.
            MyHost myHost = new MyHost(new Host02());

            Runspace myRunSpace = RunspaceFactory.CreateRunspace(myHost);
            myRunSpace.Open();

            // Create the pipeline.
```

```

Pipeline pipe = myRunSpace.CreatePipeline();

// Add the script we want to run. This script does two things.
// First, it runs the Get-Process cmdlet with the cmdlet output
// sorted by handle count. Second, the GetDate cmdlet is piped
// to the Out-String cmdlet so that we can see the
// date displayed in German.

pipe.Commands.AddScript(@"
    Get-Process | sort HandleCount
    # This should display the date in German...
    Get-Date | Out-String
");

// Add the default outputer to the end of the pipe and indicate
// that it should handle both output and errors from the previous
// commands. This will result in the output being written using the
PSHost
// and PSHostUserInterface classes instead of returning objects to the
hosting
// application.
pipe.Commands.Add("Out-Default");

pipe.Commands[0].MergeMyResults(PipelineResultTypes.Error,PipelineResultType
s.Output);

// Invoke the pipeline. There will not be any objects
// returned. The Out-Default cmdlet consumes the objects.
pipe.Invoke();

System.Console.WriteLine("Hit any key to exit...");
System.Console.ReadKey();
}

}
}

```

Example 2

The following code is the implementation of the [System.Management.Automation.Host.PSHost](#) class that is used by this host application. Those elements that are not implemented throw an exception or return nothing.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Globalization;
    using System.Management.Automation.Host;

    /// <summary>

```

```
/// This is a sample implementation of the PSHost abstract class for
/// console applications. Not all members are implemented. Those that
/// are not implemented throw a NotImplementedException exception or
/// return nothing.
/// </summary>
internal class MyHost : PSHost
{
    /// <summary>
    /// A reference to the PSHost implementation.
    /// </summary>
    private Host02 program;

    /// <summary>
    /// The culture information of the thread that created
    /// this object.
    /// </summary>
    private CultureInfo originalCultureInfo =
        System.Threading.Thread.CurrentThread.CurrentCulture;

    /// <summary>
    /// The UI culture information of the thread that created
    /// this object.
    /// </summary>
    private CultureInfo originalUICultureInfo =
        System.Threading.Thread.CurrentThread.CurrentUICulture;

    /// <summary>
    /// The identifier of this PSHost implementation.
    /// </summary>
    private Guid myId = Guid.NewGuid();

    /// <summary>
    /// Initializes a new instance of the MyHost class. Keep
    /// a reference to the host application object so that it
    /// can be informed of when to exit.
    /// </summary>
    /// <param name="program">
    /// A reference to the host application object.
    /// </param>
    public MyHost(Host02 program)
    {
        this.program = program;
    }

    /// <summary>
    /// A reference to the implementation of the PSHostUserInterface
    /// class for this application.
    /// </summary>
    private MyHostUserInterface myHostUserInterface = new
MyHostUserInterface();

    /// <summary>
    /// Gets the culture information to use. This implementation
    /// returns a snapshot of the culture information of the thread
    /// that created this object.

```

```
/// </summary>
public override System.Globalization.CultureInfo CurrentCulture
{
    get { return this.originalCultureInfo; }
}

/// <summary>
/// Gets the UI culture information to use. This implementation
/// returns a snapshot of the UI culture information of the thread
/// that created this object.
/// </summary>
public override System.Globalization.CultureInfo CurrentUICulture
{
    get { return this.originalUICultureInfo; }
}

/// <summary>
/// Gets an identifier for this host. This implementation always
/// returns the GUID allocated at instantiation time.
/// </summary>
public override Guid InstanceId
{
    get { return this.myId; }
}

/// <summary>
/// Gets a string that contains the name of this host implementation.
/// Keep in mind that this string may be used by script writers to
/// identify when your host is being used.
/// </summary>
public override string Name
{
    get { return "MySampleConsoleHostImplementation"; }
}

/// <summary>
/// Gets an instance of the implementation of the PSHostUserInterface
/// class for this application. This instance is allocated once at
startup time
/// and returned every time thereafter.
/// </summary>
public override PSHostUserInterface UI
{
    get { return this.myHostUserInterface; }
}

/// <summary>
/// Gets the version object for this application. Typically this
/// should match the version resource in the application.
/// </summary>
public override Version Version
{
    get { return new Version(1, 0, 0, 0); }
}
```

```
/// <summary>
/// This API Instructs the host to interrupt the currently running
/// pipeline and start a new nested input loop. In this example this
/// functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
public override void EnterNestedPrompt()
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API instructs the host to exit the currently running input
loop.
/// In this example this functionality is not needed so the method
/// throws a NotImplementedException exception.
/// </summary>
public override void ExitNestedPrompt()
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API is called before an external application process is
started. Typically it is used to save state so that the parent
/// can restore state that has been modified by a child process (after
/// the child exits). In this example this functionality is not
/// needed so the method returns nothing.
/// </summary>
public override void NotifyBeginApplication()
{
    return;
}

/// <summary>
/// This API is called after an external application process finishes.
/// Typically it is used to restore state that a child process has
/// altered. In this example, this functionality is not needed so
/// the method returns nothing.
/// </summary>
public override void NotifyEndApplication()
{
    return;
}

/// <summary>
/// Indicate to the host application that exit has
been requested. Pass the exit code that the host
/// application should use when exiting the process.
/// </summary>
/// <param name="exitCode">The exit code that the
/// host application should use.</param>
public override void SetShouldExit(int exitCode)
```

```

    {
        this.program.ShouldExit = true;
        this.program.ExitCode = exitCode;
    }
}
}

```

Example 3

The following code is the implementation of the [System.Management.Automation.Host.PSHostUserInterface](#) class that is used by this host application.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Collections.Generic;
    using System.Globalization;
    using System.Management.Automation;
    using System.Management.Automation.Host;

    /// <summary>
    /// A sample implementation of the PSHostUserInterface abstract class for
    /// console applications. Not all members are implemented. Those that are
    /// not implemented throw a NotImplementedException exception. Members
    /// that
    /// are implemented include those that map easily to Console APIs.
    /// </summary>
    internal class MyHostUserInterface : PSHostUserInterface
    {
        /// <summary>
        /// An instance of the PSRawUserInterface class.
        /// </summary>
        private MyRawUserInterface myRawUi = new MyRawUserInterface();

        /// <summary>
        /// Gets an instance of the PSRawUserInterface class for this host
        /// application.
        /// </summary>
        public override PSHostRawUserInterface RawUI
        {
            get { return this.myRawUi; }
        }

        /// <summary>
        /// Prompts the user for input. In this example this functionality is
        /// not
        /// needed so the method throws a NotImplementedException exception.
        /// </summary>
    }
}

```

```
/// <param name="caption">The caption or title of the prompt.</param>
/// <param name="message">The text of the prompt.</param>
/// <param name="descriptions">A collection of FieldDescription objects
that
/// describe each field of the prompt.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override Dictionary<string, PSObject> Prompt(
                                                 string caption,
                                                 string message,
System.Collections.ObjectModel.Collection<FieldDescription> descriptions)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// Provides a set of choices that enable the user to choose a
/// single option from a set of options. In this example this
/// functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="caption">Text that precedes (a title) the choices.
</param>
/// <param name="message">A message that describes the choice.</param>
/// <param name="choices">A collection of ChoiceDescription objects that
describes
/// each choice.</param>
/// <param name="defaultChoice">The index of the label in the Choices
parameter
/// collection. To indicate no default choice, set to -1.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override int PromptForChoice(string caption, string message,
System.Collections.ObjectModel.Collection<ChoiceDescription> choices, int
defaultChoice)
{
    throw new NotImplementedException("The method or operation is not
implemented.");
}

/// <summary>
/// Prompts the user for credentials with a specified prompt window
caption,
/// prompt message, user name, and target name. In this example this
/// functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="caption">The caption for the message window.</param>
/// <param name="message">The text of the message.</param>
/// <param name="userName">The user name whose credential is to be
prompted for.</param>
/// <param name="targetName">The name of the target for which the
credential is collected.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override PSCredential PromptForCredential(
```

```
        string caption,
        string message,
        string userName,
        string targetName)
    {
        throw new NotImplementedException("The method or operation is not
implemented.");
    }

    /// <summary>
    /// Prompts the user for credentials by using a specified prompt window
    /// caption,
    /// prompt message, user name and target name, credential types allowed
    /// to be
    /// returned, and UI behavior options. In this example this
    /// functionality
    /// is not needed so the method throws a NotImplementedException
    /// exception.
    /// </summary>
    /// <param name="caption">The caption for the message window.</param>
    /// <param name="message">The text of the message.</param>
    /// <param name="userName">The user name whose credential is to be
    /// prompted for.</param>
    /// <param name="targetName">The name of the target for which the
    /// credential is collected.</param>
    /// <param name="allowedCredentialTypes">A PSCredentialTypes constant
    /// that
    /// identifies the type of credentials that can be returned.</param>
    /// <param name="options">A PSCredentialUIOptions constant that
    /// identifies the UI
    /// behavior when it gathers the credentials.</param>
    /// <returns>Throws a NotImplementedException exception.</returns>
    public override PSCredential PromptForCredential(
        string caption,
        string message,
        string userName,
        string targetName,
        PSCredentialTypes
    allowedCredentialTypes,
    PSCredentialUIOptions
options)
{
    throw new NotImplementedException("The method or operation is not
implemented.");
}

    /// <summary>
    /// Reads characters that are entered by the user until a newline
    /// (carriage return) is encountered.
    /// </summary>
    /// <returns>The characters that are entered by the user.</returns>
    public override string ReadLine()
{
    return Console.ReadLine();
}
```

```
/// <summary>
/// Reads characters entered by the user until a newline (carriage
return)
/// is encountered and returns the characters as a secure string. In
this
/// example this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <returns>Throws a NotImplementedException exception.</returns>
public override System.Security.SecureString ReadLineAsSecureString()
{
    throw new NotImplementedException("The method or operation is not
implemented.");
}

/// <summary>
/// Writes characters to the output display of the host.
/// </summary>
/// <param name="value">The characters to be written.</param>
public override void Write(string value)
{
    System.Console.Write(value);
}

/// <summary>
/// Writes characters to the output display of the host and specifies
the
/// foreground and background colors of the characters. This
implementation
/// ignores the colors.
/// </summary>
/// <param name="foregroundColor">The color of the characters.</param>
/// <param name="backgroundColor">The background color to use.</param>
/// <param name="value">The characters to be written.</param>
public override void Write(
    ConsoleColor foregroundColor,
    ConsoleColor backgroundColor,
    string value)
{
    // Colors are ignored.
    System.Console.Write(value);
}

/// <summary>
/// Writes a debug message to the output display of the host.
/// </summary>
/// <param name="message">The debug message that is displayed.</param>
public override void WriteDebugLine(string message)
{
    Console.WriteLine(String.Format(
        CultureInfo.CurrentCulture,
        "DEBUG: {0}",
        message));
}
```

```
/// <summary>
/// Writes an error message to the output display of the host.
/// </summary>
/// <param name="value">The error message that is displayed.</param>
public override void WriteErrorLine(string value)
{
    Console.WriteLine(String.Format(
        CultureInfo.CurrentCulture,
        "ERROR: {0}",
        value));
}

/// <summary>
/// Writes a newline character (carriage return)
/// to the output display of the host.
/// </summary>
public override void WriteLine()
{
    System.Console.WriteLine();
}

/// <summary>
/// Writes a line of characters to the output display of the host
/// and appends a newline character(carriage return).
/// </summary>
/// <param name="value">The line to be written.</param>
public override void WriteLine(string value)
{
    System.Console.WriteLine(value);
}

/// <summary>
/// Writes a line of characters to the output display of the host
/// with foreground and background colors and appends a newline
/// (carriage return).
/// </summary>
/// <param name="foregroundColor">The foreground color of the display.
</param>
/// <param name="backgroundColor">The background color of the display.
</param>
/// <param name="value">The line to be written.</param>
public override void WriteLine(ConsoleColor ConsoleColor,
ConsoleColor ConsoleColor, string value)
{
    // Write to the output stream, ignore the colors
    System.Console.WriteLine(value);
}

/// <summary>
/// Writes a progress report to the output display of the host.
/// </summary>
/// <param name="sourceId">Unique identifier of the source of the
record. </param>
/// <param name="record">A ProgressReport object.</param>
```

```

public override void WriteProgress(long sourceId, ProgressRecord record)
{
}

/// <summary>
/// Writes a verbose message to the output display of the host.
/// </summary>
/// <param name="message">The verbose message that is displayed.</param>
public override void WriteVerboseLine(string message)
{
    Console.WriteLine(String.Format(CultureInfo.CurrentCulture, "VERBOSE:{0}", message));
}

/// <summary>
/// Writes a warning message to the output display of the host.
/// </summary>
/// <param name="message">The warning message that is displayed.</param>
public override void WriteWarningLine(string message)
{
    Console.WriteLine(String.Format(CultureInfo.CurrentCulture, "WARNING:{0}", message));
}
}

```

Example 4

The following code is the implementation of the [System.Management.Automation.Host.PSHostRawUserInterface](#) class that is used by this host application. Those elements that are not implemented throw an exception or return nothing.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Management.Automation.Host;

    /// <summary>
    /// A sample implementation of the PSHostRawUserInterface for console
    /// applications. Members of this class that easily map to the .NET
    /// console class are implemented. More complex methods are not
    /// implemented and throw a NotImplementedException exception.
    /// </summary>
    internal class MyRawUserInterface : PSHostRawUserInterface
    {
        /// <summary>
        /// Gets or sets the background color of the displayed text.
        /// This maps to the corresponding Console.BackgroundColor property.
    }
}

```

```
/// </summary>
public override ConsoleColor BackgroundColor
{
    get { return Console.BackgroundColor; }
    set { Console.BackgroundColor = value; }
}

/// <summary>
/// Gets or sets the size of the host buffer. In this example the
/// buffer size is adapted from the Console buffer size members.
/// </summary>
public override Size BufferSize
{
    get { return new Size(Console.BufferWidth, Console.BufferHeight); }
    set { Console.SetBufferSize(value.Width, value.Height); }
}

/// <summary>
/// Gets or sets the cursor position. In this example this
/// functionality is not needed so the property throws a
/// NotImplementedException exception.
/// </summary>
public override Coordinates CursorPosition
{
    get { throw new NotImplementedException(
        "The method or operation is not implemented."); }
    set { throw new NotImplementedException(
        "The method or operation is not implemented."); }
}

/// <summary>
/// Gets or sets the size of the displayed cursor. In this example
/// the cursor size is taken directly from the Console.CursorSize
/// property.
/// </summary>
public override int CursorSize
{
    get { return Console.CursorSize; }
    set { Console.CursorSize = value; }
}

/// <summary>
/// Gets or sets the foreground color of the displayed text.
/// This maps to the corresponding Console.ForegroundColor property.
/// </summary>
public override ConsoleColor ForegroundColor
{
    get { return Console.ForegroundColor; }
    set { Console.ForegroundColor = value; }
}

/// <summary>
/// Gets a value indicating whether the user has pressed a key. This
/// maps
/// to the corresponding Console.KeyAvailable property.

```

```
/// </summary>
public override bool KeyAvailable
{
    get { return Console.KeyAvailable; }
}

/// <summary>
/// Gets the dimensions of the largest window that could be
/// rendered in the current display, if the buffer was at the least
/// that large. This example uses the Console.LargestWindowSize and
/// Console.LargestWindowHeight properties to determine the returned
/// value of this property.
/// </summary>
public override Size MaxPhysicalWindowSize
{
    get { return new Size(Console.LargestWindowSize,
Console.LargestWindowHeight); }
}

/// <summary>
/// Gets the dimensions of the largest window size that can be
/// displayed. This example uses the Console.LargestWindowSize and
/// console.LargestWindowHeight properties to determine the returned
/// value of this property.
/// </summary>
public override Size MaxWindowSize
{
    get { return new Size(Console.LargestWindowSize,
Console.LargestWindowHeight); }
}

/// <summary>
/// Gets or sets the position of the displayed window. This example
/// uses the Console window position APIs to determine the returned
/// value of this property.
/// </summary>
public override Coordinates WindowPosition
{
    get { return new Coordinates(Console.WindowLeft, Console.WindowTop); }
    set { Console.SetWindowPosition(value.X, value.Y); }
}

/// <summary>
/// Gets or sets the size of the displayed window. This example
/// uses the corresponding Console window size APIs to determine the
/// returned value of this property.
/// </summary>
public override Size WindowSize
{
    get { return new Size(Console.WindowWidth, Console.WindowHeight); }
    set { Console.SetWindowSize(value.Width, value.Height); }
}

/// <summary>
/// Gets or sets the title of the displayed window. The example
```

```
/// maps the Console.Title property to the value of this property.
/// </summary>
public override string WindowTitle
{
    get { return Console.Title; }
    set { Console.Title = value; }
}

/// <summary>
/// This API resets the input buffer. In this example this
/// functionality is not needed so the method returns nothing.
/// </summary>
public override void FlushInputBuffer()
{
}

/// <summary>
/// This API returns a rectangular region of the screen buffer. In
/// this example this functionality is not needed so the method throws
/// a NotImplementedException exception.
/// </summary>
/// <param name="rectangle">Defines the size of the rectangle.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override BufferCell[,] GetBufferContents(Rectangle rectangle)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API reads a pressed, released, or pressed and released
keystroke
    /// from the keyboard device, blocking processing until a keystroke is
    /// typed that matches the specified keystroke options. In this example
    /// this functionality is not needed so the method throws a
    /// NotImplementedException exception.
    /// </summary>
    /// <param name="options">Options, such as IncludeKeyDown, used when
    /// reading the keyboard.</param>
    /// <returns>Throws a NotImplementedException exception.</returns>
public override KeyInfo ReadKey(ReadKeyOptions options)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API crops a region of the screen buffer. In this example
/// this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="source">The region of the screen to be scrolled.
</param>
    /// <param name="destination">The region of the screen to receive the
    /// source region contents.</param>
```

```

    /// <param name="clip">The region of the screen to include in the
    operation.</param>
    /// <param name="fill">The character and attributes to be used to fill
    all cell.</param>
    public override void ScrollBufferContents(Rectangle source, Coordinates
destination, Rectangle clip, BufferCell fill)
    {
        throw new NotImplementedException(
            "The method or operation is not implemented.");
    }

    /// <summary>
    /// This method copies an array of buffer cells into the screen buffer
    /// at a specified location. In this example this functionality is
    /// not needed so the method throws a NotImplementedException exception.
    /// </summary>
    /// <param name="origin">The parameter is not used.</param>
    /// <param name="contents">The parameter is not used.</param>
    public override void SetBufferContents(Coordinates origin,
                                         BufferCell[,] contents)
    {
        throw new NotImplementedException(
            "The method or operation is not implemented.");
    }

    /// <summary>
    /// This method copies a given character, foreground color, and
background
    /// color to a region of the screen buffer. In this example this
    /// functionality is not needed so the method throws a
    /// NotImplementedException exception./// </summary>
    /// <param name="rectangle">Defines the area to be filled. </param>
    /// <param name="fill">Defines the fill character.</param>
    public override void SetBufferContents(Rectangle rectangle, BufferCell
fill)
    {
        throw new NotImplementedException(
            "The method or operation is not implemented.");
    }
}

```

See Also

[System.Management.Automation.PowerShell](#)

[System.Management.Automation.Host.PSHost](#)

[System.Management.Automation.Host.PSHostUserInterface](#)

[System.Management.Automation.Host.PSHostRawUserInterface](#)

Host03 Sample

Article • 09/17/2021

This sample shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

- Creating a custom host whose classes derive from the [System.Management.Automation.Host.PSHost](#) class, the [System.Management.Automation.Host.PSHostUserInterface](#) class, and the [System.Management.Automation.Host.PSHostRawUserInterface](#) class.
- Building a console application that uses these host classes to build an interactive Windows PowerShell shell.

Example 1

This example allows the user to enter commands at a command line, processes those commands, and then prints out the results.

C#

```
// Copyright (c) 2006 Microsoft Corporation. All rights reserved.  
//  
// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF  
// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO  
// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A  
// PARTICULAR PURPOSE.  
  
namespace Microsoft.Samples.PowerShell.Host  
{  
    using System;  
    using System.Collections.ObjectModel;  
    using System.Management.Automation;  
    using System.Management.Automation.Runspaces;  
    using PowerShell = System.Management.Automation.PowerShell;
```

```
/// This class contains the Main entry point for this host application.
internal class PSListenerConsoleSample
{
    /// Indicator to tell the host application that it should exit.
    private bool shouldExit;

    /// The exit code that the host application will use to exit.
    private int exitCode;

    /// Holds the instance of the PSHost implementation for this
    interpreter.
    private MyHost myHost;

    /// Holds the runspace for this interpreter.
    private Runspace myRunSpace;

    /// Holds a reference to the currently executing pipeline so it can be
    /// stopped by the control-C handler.
    private PowerShell currentPowerShell;

    /// Used to serialize access to instance data.
    private object instanceLock = new object();

    /// Create this instance of the console listener.
    private PSListenerConsoleSample()
    {
        // Create the host and runspace instances for this interpreter.
        // Note that this application does not support console files so
        // only the default snapins will be available.
        this.myHost = new MyHost(this);
        this.myRunSpace = RunspaceFactory.CreateRunspace(this.myHost);
        this.myRunSpace.Open();
    }

    /// Gets or sets a value indicating whether the host application
    /// should exit.
    public bool ShouldExit
    {
        get { return this.shouldExit; }
        set { this.shouldExit = value; }
    }

    /// Gets or sets the exit code that the host application will use
    /// when exiting.
    public int ExitCode
    {
        get { return this.exitCode; }
        set { this.exitCode = value; }
    }

    /// Creates and initiates the listener instance.
    /// param name="args";This parameter is not used.
    private static void Main(string[] args)
    {
```

```

// Display the welcome message...
Console.Title = "PowerShell Console Host Sample Application";
ConsoleColor oldFg = Console.ForegroundColor;
Console.ForegroundColor = ConsoleColor.Cyan;
Console.WriteLine("    PowerShell Console Host Interactive Sample");
Console.WriteLine("    ======");
Console.WriteLine(string.Empty);
Console.WriteLine("This is an example of a simple interactive console
host that uses the ");
Console.WriteLine("Windows PowerShell engine to interpret commands.
Type 'exit' to exit.");
Console.WriteLine(string.Empty);
Console.ForegroundColor = oldFg;

// Create the listener and run it - this never returns...
PSListenerConsoleSample listener = new PSListenerConsoleSample();
listener.Run();
}

/// A helper class that builds and executes a pipeline that writes to
the
/// default output path. Any exceptions that are thrown are just passed
to
/// the caller. Since all output goes to the default outputter, this
method()
/// won't return anything.
/// param name="cmd"; The script to run.
/// param name="input";Any input arguments to pass to the script. If
null
/// then nothing is passed in.
private void executeHelper(string cmd, object input)
{
    // Ignore empty command lines.
    if (String.IsNullOrEmpty(cmd))
    {
        return;
    }

    // Create the pipeline object and make it available
    // to the ctrl-C handle through the currentPowerShell instance
    // variable
    lock (this.instanceLock)
    {
        this.currentPowerShell = PowerShell.Create();
    }

    this.currentPowerShell.Runspace = this.myRunSpace;

    // Create a pipeline for this execution. Place the result in the
    // currentPowerShell instance variable so that it is available
    // to be stopped.
    try
    {
        this.currentPowerShell.AddScript(cmd);
    }
}

```

```

        // Now add the default outputter to the end of the pipe and indicate
        // that it should handle both output and errors from the previous
        // commands. This will result in the output being written using the
    PSHost
        // and PSHostUserInterface classes instead of returning objects to
        // the hosting
        // application.
        this.currentPowerShell.AddCommand("Out-Default");

this.currentPowerShell.Commands.Commands[0].MergeMyResults(PipelineResultTyp
es.Error, PipelineResultTypes.Output);

        // If there was any input specified, pass it in, otherwise just
        // execute the pipeline.
        if (input != null)
        {
            this.currentPowerShell.Invoke(new object[] { input });
        }
        else
        {
            this.currentPowerShell.Invoke();
        }
    }
    finally
    {
        // Dispose of the pipeline line and set it to null, locked because
        // currentPowerShell may be accessed by the ctrl-C handler.
        lock (this.instanceLock)
        {
            this.currentPowerShell.Dispose();
            this.currentPowerShell = null;
        }
    }
}

/// An exception occurred that we want to display
/// using the display formatter. To do this we run
/// a second pipeline passing in the error record.
/// The runtime will bind this to the $input variable
/// which is why $input is being piped to Out-String.
/// We then call WriteErrorLine to make sure the error
/// gets displayed in the correct error color.

/// param name="e"; The exception to display.
private void ReportException(Exception e)
{
    if (e != null)
    {
        object error;
        IContainsErrorRecord icer = e as IContainsErrorRecord;
        if (icer != null)
        {
            error = icer.ErrorRecord;
        }
        else

```

```

        {
            error = (object) new ErrorRecord(e, "Host.ReportException",
ErrorCategory.NotSpecified, null);
        }

        lock (this.instanceLock)
{
    this.currentPowerShell = PowerShell.Create();
}

this.currentPowerShell.Runspace = this.myRunSpace;

try
{
    this.currentPowerShell.AddScript("$input").AddCommand("Out-
String");

        // Do not merge errors, this function will swallow errors.
Collection<PSObject> result;
PSDataCollection<object> inputCollection = new
PSDataCollection<object>();
inputCollection.Add(error);
inputCollection.Complete();
result = this.currentPowerShell.Invoke(inputCollection);

if (result.Count > 0)
{
    string str = result[0].BaseObject as string;
    if (!string.IsNullOrEmpty(str))
    {
        // Remove \r\n that is added by Out-String.
        this.myHost.UI.WriteLine(str.Substring(0, str.Length -
2));
    }
}
finally
{
    // Dispose of the pipeline line and set it to null, locked because
currentPowerShell
        // may be accessed by the ctrl-C handler.
lock (this.instanceLock)
{
    this.currentPowerShell.Dispose();
    this.currentPowerShell = null;
}
}
}

/// Basic script execution routine - any runtime exceptions are
/// caught and passed back into the engine to display.

/// param name="cmd"; The parameter is not used.
private void Execute(string cmd)

```

```

{
    try
    {
        // Execute the command with no input.
        this.executeHelper(cmd, null);
    }
    catch (RuntimeException rte)
    {
        this.ReportException(rte);
    }
}

/// Method used to handle control-C's from the user. It calls the
/// pipeline Stop() method to stop execution. If any exceptions occur
/// they are printed to the console but otherwise ignored.

/// param name="sender"; See sender property of
ConsoleCancelEventHandler documentation.
/// param name="e"; See e property of ConsoleCancelEventArgs
documentation.
private void HandleControlC(object sender, ConsoleCancelEventArgs e)
{
    try
    {
        lock (this.instanceLock)
        {
            if (this.currentPowerShell != null &&
this.currentPowerShell.InvocationStateInfo.State ==
PSInvocationState.Running)
            {
                this.currentPowerShell.Stop();
            }
        }

        e.Cancel = true;
    }
    catch (Exception exception)
    {
        this.myHost.UI.WriteLine(exception.ToString());
    }
}

/// Implements the basic listener loop. It sets up the ctrl-C handler,
then
/// reads a command from the user, executes it and repeats until the
ShouldExit
/// flag is set.
private void Run()
{
    // Set up the control-C handler.
    Console.CancelKeyPress += new
ConsoleCancelEventHandler(this.HandleControlC);
    Console.TreatControlCAsInput = false;

    // Read commands to execute until ShouldExit is set by

```

```

        // the user calling "exit".
        while (!this.ShouldExit)
        {
            this.myHost.UI.WriteLine(ConsoleColor.Cyan, ConsoleColor.Black,
"\nPSConsoleSample: ");
            string cmd = Console.ReadLine();
            this.Execute(cmd);
        }

        // Exit with the desired exit code that was set by exit command.
        // This is set in the host by the MyHost.SetShouldExit()
implementation.
        Environment.Exit(this.ExitCode);
    }
}
}

```

Example 2

The following code is the implementation of the [System.Management.Automation.Host.PSHost](#) class that is used by this host application. Those elements that are not implemented throw an exception or return nothing.

C#

```

// Copyright (c) 2006 Microsoft Corporation. All rights reserved.
//
// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
// PARTICULAR PURPOSE.
//
using System;
using System.Collections.Generic;
using System.Text;
using System.Management.Automation;
using System.Management.Automation.Host;
using System.Management.Automation.Runspaces;

namespace Microsoft.Samples.PowerShell.Host
{
    /// <summary>
    /// Simple PowerShell interactive console host listener implementation.
    /// This class
    /// implements a basic read-evaluate-print loop or 'listener' allowing you
    /// to
    /// interactively work with the PowerShell runtime.
    /// </summary>
    class PSListenerConsoleSample
    {
        /// <summary>

```

```
    /// Define the property that the PSHost implementation will use to tell
    /// the host
    /// application that it should exit.
    /// </summary>
    public bool ShouldExit
    {
        get { return shouldExit; }
        set { shouldExit = value; }
    }
    private bool shouldExit;

    /// <summary>
    /// Define the property that the PSHost implementation will use to tell
    /// the host
    /// application what code to use when exiting.
    /// </summary>
    public int ExitCode
    {
        get { return exitCode; }
        set { exitCode = value; }
    }
    private int exitCode;
    /// <summary>
    /// Holds the instance of the PSHost implementation for this
    /// interpreter.
    /// </summary>
    private MyHost myHost;

    /// <summary>
    /// Holds the runspace for this interpreter.
    /// </summary>
    private Runspace myRunSpace;

    /// <summary>
    /// Holds a reference to the currently executing pipeline so it can be
    /// stopped by the control-C handler.
    /// </summary>
    private Pipeline currentPipeline;

    /// <summary>
    /// Used to serialize access to instance data...
    /// </summary>
    private object instanceLock = new object();

    /// <summary>
    /// Create this instance of the console listener.
    /// </summary>
    PSListenerConsoleSample()
    {
        // Create the host and runspace instances for this interpreter. Note
        // that
        // this application doesn't support console files so only the default
        // snapins
        // will be available.
        myHost = new MyHost(this);
    }
}
```

```

        myRunSpace = RunspaceFactory.CreateRunspace(myHost);
        myRunSpace.Open();
    }

    /// <summary>
    /// A helper class that builds and executes a pipeline that writes to
    the
    /// default output path. Any exceptions that are thrown are just passed
    to
    /// the caller. Since all output goes to the default outputter, this
    method()
    /// won't return anything.
    /// </summary>
    /// <param name="cmd">The script to run</param>
    /// <param name="input">Any input arguments to pass to the script. If
    null
    /// then nothing is passed in.</param>
    void executeHelper(string cmd, object input)
    {
        // Ignore empty command lines.
        if (String.IsNullOrEmpty(cmd))
            return;

        // Create the pipeline object and make it available
        // to the ctrl-C handle through the currentPipeline instance
        // variable.
        lock (instanceLock)
        {
            currentPipeline = myRunSpace.CreatePipeline();
        }

        // Create a pipeline for this execution. Place the result in the
        currentPipeline
        // instance variable so that it is available to be stopped.
        try
        {
            currentPipeline.Commands.AddScript(cmd);

            // Now add the default outputter to the end of the pipe and indicate
            // that it should handle both output and errors from the previous
            // commands. This will result in the output being written using the
            PSHost
            // and PSHostUserInterface classes instead of returning objects to
            the hosting
            // application.
            currentPipeline.Commands.Add("Out-Default");

            currentPipeline.Commands[0].MergeMyResults(PipelineResultTypes.Error,
            PipelineResultTypes.Output);

            // If there was any input specified, pass it in, otherwise just
            // execute the pipeline.
            if (input != null)
            {
                currentPipeline.Invoke(new object[] { input });
            }
        }
    }
}

```

```

        }
        else
        {
            currentPipeline.Invoke();
        }
    }
    finally
    {
        // Dispose of the pipeline line and set it to null, locked because
        currentPipeline
        // may be accessed by the ctrl-C handler.
        lock (instanceLock)
        {
            currentPipeline.Dispose();
            currentPipeline = null;
        }
    }
}

/// <summary>
/// Basic script execution routine - any runtime exceptions are
/// caught and passed back into the runtime to display.
/// </summary>
/// <param name="cmd"></param>
void Execute(string cmd)
{
    try
    {
        // execute the command with no input...
        executeHelper(cmd, null);
    }
    catch (RuntimeException rte)
    {
        // An exception occurred that we want to display
        // using the display formatter. To do this we run
        // a second pipeline passing in the error record.
        // The runtime will bind this to the $input variable
        // which is why $input is being piped to Out-Default
        executeHelper("$input | Out-Default", rte.ErrorRecord);
    }
}

/// <summary>
/// Method used to handle control-C's from the user. It calls the
/// pipeline Stop() method to stop execution. If any exceptions occur,
/// they are printed to the console; otherwise they are ignored.
/// </summary>
/// <param name="sender">See ConsoleCancelEventHandler
documentation</param>
/// <param name="e">See ConsoleCancelEventHandler documentation</param>
void HandleControlC(object sender, ConsoleCancelEventArgs e)
{
    try
    {
        lock (instanceLock)

```

```

    {
        if (currentPipeline != null &&
currentPipeline.PipelineStateInfo.State == PipelineState.Running)
            currentPipeline.Stop();
    }
    e.Cancel = true;
}
catch (Exception exception)
{
    this.myHost.UI.WriteLine(exception.ToString());
}
}

/// <summary>
/// Implements the basic listener loop. It sets up the ctrl-C handler,
then
/// reads a command from the user, executes it and repeats until the
ShouldExit
/// flag is set.
/// </summary>
private void Run()
{
    // Set up the control-C handler.
    Console.KeyPress += new
ConsoleCancelEventHandler(HandleControlC);
    Console.TreatControlCAsInput = false;

    // Loop reading commands to execute until ShouldExit is set by
    // the user calling "exit".
    while (!ShouldExit)
    {
        myHost.UI.Write(ConsoleColor.Cyan, ConsoleColor.Black,
"\nPSConsoleSample: ");
        string cmd = Console.ReadLine();
        Execute(cmd);
    }

    // Exit with the desired exit code that was set by exit command.
    // This is set in the host by the MyHost.SetShouldExit()
implementation.
    Environment.Exit(ExitCode);
}

/// <summary>
/// Creates and initiates the listener instance.
/// </summary>
/// <param name="args">Ignored for now.</param>
static void Main(string[] args)
{
    // Display the welcome message.
    Console.Title = "PowerShell Console Host Sample Application";
    ConsoleColor oldFg = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine("    PowerShell Console Host Interactive Sample");
    Console.WriteLine("    =====");
}

```

```

        Console.WriteLine("");
        Console.WriteLine("This is an example of a simple interactive console
host using the PowerShell");
        Console.WriteLine("engine to interpret commands. Type 'exit' to
exit.");
        Console.WriteLine("");
        Console.ForegroundColor = oldFg;

        // Create the listener and run it - this never returns.
        PSListenerConsoleSample listener = new PSListenerConsoleSample();
        listener.Run();
    }
}
}

```

Example 3

The following code is the implementation of the [System.Management.Automation.Host.PSHostUserInterface](#) class that is used by this host application.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Globalization;
    using System.Management.Automation;
    using System.Management.Automation.Host;
    using System.Text;

    /// <summary>
    /// A sample implementation of the PSHostUserInterface abstract class for
    /// console applications. Not all members are implemented. Those that are
    /// not implemented throw a NotImplementedException exception or return
    /// nothing. Members that are implemented include those that map easily to
    /// Console APIs and a basic implementation of the prompt API provided.
    /// </summary>
    internal class MyHostUserInterface : PSHostUserInterface
    {
        /// <summary>
        /// An instance of the PSRawUserInterface object.
        /// </summary>
        private MyRawUserInterface myRawUi = new MyRawUserInterface();

        /// <summary>
        /// Gets an instance of the PSRawUserInterface object for this host
        /// application.
        /// </summary>

```

```

public override PSHostRawUserInterface RawUI
{
    get { return this.myRawUi; }
}

/// <summary>
/// Prompts the user for input.
/// <param name="caption">The caption or title of the prompt.</param>
/// <param name="message">The text of the prompt.</param>
/// <param name="descriptions">A collection of FieldDescription objects
/// that
/// describe each field of the prompt.</param>
/// <returns>A dictionary object that contains the results of the user
/// prompts.</returns>
public override Dictionary<string, PSObject> Prompt(
    string caption,
    string message,
    Collection<FieldDescription>
descriptions)
{
    this.WriteLine(
        ConsoleColor.Blue,
        ConsoleColor.Black,
        caption + "\n" + message + " ");
    Dictionary<string, PSObject> results =
        new Dictionary<string, PSObject>();
    foreach (FieldDescription fd in descriptions)
    {
        string[] label = GetHotkeyAndLabel(fd.Label);
        this.WriteLine(label[1]);
        string userData = Console.ReadLine();
        if (userData == null)
        {
            return null;
        }

        results[fd.Name] = PSObject.AsPSObject(userData);
    }

    return results;
}

/// <summary>

/// Provides a set of choices that enable the user to choose a
/// single option from a set of options.
/// </summary>
/// <param name="caption">Text that proceeds (a title) the choices.
</param>
/// <param name="message">A message that describes the choice.</param>
/// <param name="choices">A collection of ChoiceDescription objects that
/// describe
/// each choice.</param>
/// <param name="defaultChoice">The index of the label in the Choices
/// parameter

```

```

/// collection. To indicate no default choice, set to -1.</param>
/// <returns>The index of the Choices parameter collection element that
corresponds
/// to the option that is selected by the user.</returns>
public override int PromptForChoice(
    string caption,
    string message,
    Collection<ChoiceDescription>
choices,
    int defaultChoice)
{
    // Write the caption and message strings in Blue.
    this.WriteLine(
        ConsoleColor.Blue,
        ConsoleColor.Black,
        caption + "\n" + message + "\n");

    // Convert the choice collection into something that is easier to
    // work with. See the BuildHotkeysAndPlainLabels method for details.
    Dictionary<string, PSObject> results =
        new Dictionary<string, PSObject>();
    string[,] promptData = BuildHotkeysAndPlainLabels(choices);

    // Format the overall choice prompt string to display...
    StringBuilder sb = new StringBuilder();
    for (int element = 0; element < choices.Count; element++)
    {
        sb.Append(String.Format(
            CultureInfo.CurrentCulture,
            "|{0} > {1} ",
            promptData[0, element],
            promptData[1, element]));
    }

    sb.Append(String.Format(
        CultureInfo.CurrentCulture,
        "[Default is ({0})",
        promptData[0, defaultChoice]));

    // Read prompts until a match is made, the default is
    // chosen, or the loop is interrupted with ctrl-C.
    while (true)
    {
        this.WriteLine(ConsoleColor.Cyan, ConsoleColor.Black,
sb.ToString());
        string data =
Console.ReadLine().Trim().ToUpper(CultureInfo.CurrentCulture);

        // If the choice string was empty, use the default selection.
        if (data.Length == 0)
        {
            return defaultChoice;
        }

        // See if the selection matched and return the

```

```

        // corresponding index if it did.
        for (int i = 0; i < choices.Count; i++)
        {
            if (promptData[0, i] == data)
            {
                return i;
            }
        }

        this.WriteLine("Invalid choice: " + data);
    }
}

/// <summary>
/// Prompts the user for credentials with a specified prompt window
caption,
/// prompt message, user name, and target name. In this example this
/// functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="caption">The caption for the message window.</param>
/// <param name="message">The text of the message.</param>
/// <param name="userName">The user name whose credential is to be
prompted for.</param>
/// <param name="targetName">The name of the target for which the
credential is collected.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override PSCredential PromptForCredential(
    string caption,
    string message,
    string userName,
    string targetName)
{
    throw new NotImplementedException("The method or operation is not
implemented.");
}

/// <summary>
/// Prompts the user for credentials by using a specified prompt window
caption,
/// prompt message, user name and target name, credential types allowed
to be
/// returned, and UI behavior options. In this example this
functionality
/// is not needed so the method throws a NotImplementedException
exception.
/// </summary>
/// <param name="caption">The caption for the message window.</param>
/// <param name="message">The text of the message.</param>
/// <param name="userName">The user name whose credential is to be
prompted for.</param>
/// <param name="targetName">The name of the target for which the
credential is collected.</param>
/// <param name="allowedCredentialTypes">A PSCredentialTypes constant
that

```

```
    /// identifies the type of credentials that can be returned.</param>
    /// <param name="options">A PSCredentialUIOptions constant that
    identifies the UI
    /// behavior when it gathers the credentials.</param>
    /// <returns>Throws a NotImplementedException exception.</returns>
    public override PSCredential PromptForCredential(
        string caption,
        string message,
        string userName,
        string targetName,
        PSCredentialTypes
    allowedCredentialTypes,
    PSCredentialUIOptions
options)
{
    throw new NotImplementedException("The method or operation is not
implemented.");
}

/// <summary>
/// Reads characters that are entered by the user until a newline
/// (carriage return) is encountered.
/// </summary>
/// <returns>The characters that are entered by the user.</returns>
public override string ReadLine()
{
    return Console.ReadLine();
}

/// <summary>
/// Reads characters entered by the user until a newline (carriage
return)
/// is encountered and returns the characters as a secure string. In
this
/// example this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <returns>Throws a NotImplementedException exception.</returns>
public override System.Security.SecureString ReadLineAsSecureString()
{
    throw new NotImplementedException("The method or operation is not
implemented.");
}

/// <summary>
/// Writes characters to the output display of the host.
/// </summary>
/// <param name="value">The characters to be written.</param>
public override void Write(string value)
{
    Console.Write(value);
}

/// <summary>
/// Writes characters to the output display of the host with possible
```

```
/// foreground and background colors.
/// </summary>
/// <param name="foregroundColor">The color of the characters.</param>
/// <param name="backgroundColor">The background color to use.</param>
/// <param name="value">The characters to be written.</param>
public override void Write(
    ConsoleColor foregroundColor,
    ConsoleColor backgroundColor,
    string value)
{
    ConsoleColor oldFg = Console.ForegroundColor;
    ConsoleColor oldBg = Console.BackgroundColor;
    Console.ForegroundColor = foregroundColor;
    Console.BackgroundColor = backgroundColor;
    Console.Write(value);
    Console.ForegroundColor = oldFg;
    Console.BackgroundColor = oldBg;
}

/// <summary>
/// Writes a line of characters to the output display of the host
/// with foreground and background colors and appends a newline
/// (carriage return).
/// </summary>
/// <param name="foregroundColor">The foreground color of the display.
</param>
/// <param name="backgroundColor">The background color of the display.
</param>
/// <param name="value">The line to be written.</param>
public override void WriteLine(
    ConsoleColor foregroundColor,
    ConsoleColor backgroundColor,
    string value)
{
    ConsoleColor oldFg = Console.ForegroundColor;
    ConsoleColor oldBg = Console.BackgroundColor;
    Console.ForegroundColor = foregroundColor;
    Console.BackgroundColor = backgroundColor;
    Console.WriteLine(value);
    Console.ForegroundColor = oldFg;
    Console.BackgroundColor = oldBg;
}

/// <summary>
/// Writes a debug message to the output display of the host.
/// </summary>
/// <param name="message">The debug message that is displayed.</param>
public override void WriteDebugLine(string message)
{
    this.WriteLine(
        ConsoleColor.DarkYellow,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "DEBUG: {0}",
message));
}
```

```
/// <summary>
/// Writes an error message to the output display of the host.
/// </summary>
/// <param name="value">The error message that is displayed.</param>
public override void WriteErrorLine(string value)
{
    this.WriteLine(
        ConsoleColor.Red,
        ConsoleColor.Black,
        value);
}

/// <summary>
/// Writes a newline character (carriage return)
/// to the output display of the host.
/// </summary>
public override void WriteLine()
{
    Console.WriteLine();
}

/// <summary>
/// Writes a line of characters to the output display of the host
/// and appends a newline character(carriage return).
/// </summary>
/// <param name="value">The line to be written.</param>
public override void WriteLine(string value)
{
    Console.WriteLine(value);
}

/// <summary>
/// Writes a progress report to the output display of the host.
/// </summary>
/// <param name="sourceId">Unique identifier of the source of the
record. </param>
/// <param name="record">A ProgressReport object.</param>
public override void WriteProgress(long sourceId, ProgressRecord record)
{

}

/// <summary>
/// Writes a verbose message to the output display of the host.
/// </summary>
/// <param name="message">The verbose message that is displayed.</param>
public override void WriteVerboseLine(string message)
{
    this.WriteLine(
        ConsoleColor.Green,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "VERBOSE:
{0}", message));
}
```

```

/// <summary>
/// Writes a warning message to the output display of the host.
/// </summary>
/// <param name="message">The warning message that is displayed.</param>
public override void WriteWarningLine(string message)
{
    this.WriteLine(
        ConsoleColor.Yellow,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "WARNING:
{0}", message));
}

/// <summary>
/// This is a private worker function splits out the
/// accelerator keys from the menu and builds a two
/// dimensional array with the first access containing the
/// accelerator and the second containing the label string
/// with the & removed.
/// </summary>
/// <param name="choices">The choice collection to process</param>
/// <returns>
/// A two dimensional array containing the accelerator characters
/// and the cleaned-up labels</returns>
private static string[,] BuildHotkeysAndPlainLabels(
    Collection<ChoiceDescription> choices)
{
    // Allocate the result array.
    string[,] hotkeysAndPlainLabels = new string[2, choices.Count];
    for (int i = 0; i < choices.Count; ++i)
    {
        string[] hotkeyAndLabel = GetHotkeyAndLabel(choices[i].Label);
        hotkeysAndPlainLabels[0, i] = hotkeyAndLabel[0];
        hotkeysAndPlainLabels[1, i] = hotkeyAndLabel[1];
    }

    return hotkeysAndPlainLabels;
}

/// <summary>
/// Parse a string containing a hotkey character.
/// Take a string of the form
///     Yes to &all
/// and returns a two-dimensional array split out as
///     "A", "Yes to all".
/// </summary>
/// <param name="input">The string to process</param>
/// <returns>
/// A two dimensional array containing the parsed components.
/// </returns>
private static string[] GetHotkeyAndLabel(string input)
{
    string[] result = new string[] { String.Empty, String.Empty };
    string[] fragments = input.Split('&');

```

```

        if (fragments.Length == 2)
    {
        if (fragments[1].Length > 0)
        {
            result[0] = fragments[1][0].ToString().
                ToUpper(CultureInfo.CurrentCulture);
        }

        result[1] = (fragments[0] + fragments[1]).Trim();
    }
    else
    {
        result[1] = input;
    }

    return result;
}
}
}

```

Example 4

The following code is the implementation of the [System.Management.Automation.Host.PSHostRawUserInterface](#) class that is used by this host application. Those elements that are not implemented throw an exception or return nothing.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Management.Automation.Host;

    /// <summary>
    /// A sample implementation of the PSHostRawUserInterface for console
    /// applications. Members of this class that easily map to the .NET
    /// console class are implemented. More complex methods are not
    /// implemented and throw a NotImplementedException exception.
    /// </summary>
    internal class MyRawUserInterface : PSHostRawUserInterface
    {
        /// <summary>
        /// Gets or sets the background color of text to be written.
        /// This maps to the corresponding Console.BackgroundColor property.
        /// </summary>
        public override ConsoleColor BackgroundColor
        {
            get { return Console.BackgroundColor; }
            set { Console.BackgroundColor = value; }
        }
    }
}

```

```
/// <summary>
/// Gets or sets the host buffer size adapted from the Console buffer
/// size members.
/// </summary>
public override Size BufferSize
{
    get { return new Size(Console.BufferWidth, Console.BufferHeight); }
    set { Console.SetBufferSize(value.Width, value.Height); }
}

/// <summary>
/// Gets or sets the cursor position. In this example this
/// functionality is not needed so the property throws a
/// NotImplementedException exception.
/// </summary>
public override Coordinates CursorPosition
{
    get { throw new NotImplementedException(
        "The method or operation is not implemented."); }
    set { throw new NotImplementedException(
        "The method or operation is not implemented."); }
}

/// <summary>
/// Gets or sets the cursor size taken directly from the
/// Console.CursorSize property.
/// </summary>
public override int CursorSize
{
    get { return Console.CursorSize; }
    set { Console.CursorSize = value; }
}

/// <summary>
/// Gets or sets the foreground color of the text to be written.
/// This maps to the corresponding Console.ForegroundColor property.
/// </summary>
public override ConsoleColor ForegroundColor
{
    get { return Console.ForegroundColor; }
    set { Console.ForegroundColor = value; }
}

/// <summary>
/// Gets a value indicating whether a key is available. This maps to
/// the corresponding Console.KeyAvailable property.
/// </summary>
public override bool KeyAvailable
{
    get { return Console.KeyAvailable; }
}

/// <summary>
/// Gets the maximum physical size of the window adapted from the
```

```
///  Console.LargestWindowWidth and Console.LargestWindowSize
///  properties.
/// </summary>
public override Size MaxPhysicalWindowSize
{
    get { return new Size(Console.LargestWindowWidth,
Console.LargestWindowSize); }
}

/// <summary>
/// Gets the maximum window size adapted from the
/// Console.LargestWindowWidth and console.LargestWindowSize
/// properties.
/// </summary>
public override Size MaxWindowSize
{
    get { return new Size(Console.LargestWindowWidth,
Console.LargestWindowSize); }
}

/// <summary>
/// Gets or sets the window position adapted from the Console window
position
/// members.
/// </summary>
public override Coordinates WindowPosition
{
    get { return new Coordinates(Console.WindowLeft, Console.WindowTop); }
    set { Console.SetWindowPosition(value.X, value.Y); }
}

/// <summary>
/// Gets or sets the window size adapted from the corresponding Console
/// calls.
/// </summary>
public override Size WindowSize
{
    get { return new Size(Console.WindowWidth, Console.WindowHeight); }
    set { Console.SetWindowSize(value.Width, value.Height); }
}

/// <summary>
/// Gets or sets the title of the window mapped to the Console.Title
/// property.
/// </summary>
public override string WindowTitle
{
    get { return Console.Title; }
    set { Console.Title = value; }
}

/// <summary>
/// This API resets the input buffer. In this example this
/// functionality is not needed so the method returns nothing.
/// </summary>
```

```
public override void FlushInputBuffer()
{
}

/// <summary>
/// This API returns a rectangular region of the screen buffer. In
/// this example this functionality is not needed so the method throws
/// a NotImplementedException exception.
/// </summary>
/// <param name="rectangle">Defines the size of the rectangle.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override BufferCell[,] GetBufferContents(Rectangle rectangle)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API Reads a pressed, released, or pressed and released
keystroke
/// from the keyboard device, blocking processing until a keystroke is
/// typed that matches the specified keystroke options. In this example
/// this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="options">Options, such as IncludeKeyDown, used when
/// reading the keyboard.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override KeyInfo ReadKey(ReadKeyOptions options)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API crops a region of the screen buffer. In this example
/// this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="source">The region of the screen to be scrolled.
</param>
/// <param name="destination">The region of the screen to receive the
/// source region contents.</param>
/// <param name="clip">The region of the screen to include in the
operation.</param>
/// <param name="fill">The character and attributes to be used to fill
all cell.</param>
public override void ScrollBufferContents(Rectangle source, Coordinates
destination, Rectangle clip, BufferCell fill)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
```

```
    /// This API copies an array of buffer cells into the screen buffer
    /// at a specified location. In this example this functionality is
    /// not needed so the method throws a NotImplementedException
    exception.
    /// </summary>
    /// <param name="origin">The parameter is not used.</param>
    /// <param name="contents">The parameter is not used.</param>
    public override void SetBufferContents(Coordinates origin, BufferCell[,] contents)
    {
        throw new NotImplementedException(
            "The method or operation is not implemented.");
    }

    /// <summary>
    /// This API Copies a given character, foreground color, and background
    /// color to a region of the screen buffer. In this example this
    /// functionality is not needed so the method throws a
    /// NotImplementedException exception./// </summary>
    /// <param name="rectangle">Defines the area to be filled. </param>
    /// <param name="fill">Defines the fill character.</param>
    public override void SetBufferContents(Rectangle rectangle, BufferCell fill)
    {
        throw new NotImplementedException(
            "The method or operation is not implemented.");
    }
}
```

See Also

[System.Management.Automation.Host.PSHost](#)

[System.Management.Automation.Host.PSHostUserInterface](#)

[System.Management.Automation.Host.PSHostRawUserInterface](#)

Host04 Sample

Article • 03/24/2025

This sample shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console. This host application also supports displaying prompts that allow the user to specify multiple choices.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

- Creating a custom host whose classes derive from the [System.Management.Automation.Host.PSHost](#) class, the [System.Management.Automation.Host.PSHostUserInterface](#) class, and the [System.Management.Automation.Host.PSHostRawUserInterface](#) class.
- Building a console application that uses these host classes to build an interactive Windows PowerShell shell.
- Creating a `$PROFILE` variable and loading the following profiles.
 - current user, current host
 - current user, all hosts
 - all users, current host
 - all users, all hosts
- Implement the [System.Management.Automation.Host.IHostUISupportsMultipleChoiceSelection](#) interface.

Example 1

This example allows the user to enter commands at a command line, processes those commands, and then prints out the results.

```
C#
```

```
namespace Microsoft.Samples.PowerShell.Host
{
```

```
using System;
using System.Collections.ObjectModel;
using System.Management.Automation;
using System.Management.Automation.Runspaces;
using PowerShell = System.Management.Automation.PowerShell;

/// <summary>
/// This class contains the Main entry point for this host application.
/// </summary>
internal class PSListenerConsoleSample
{
    /// <summary>
    /// Indicator to tell the host application that it should exit.
    /// </summary>
    private bool shouldExit;

    /// <summary>
    /// The exit code that the host application will use to exit.
    /// </summary>
    private int exitCode;

    /// <summary>
    /// Holds a reference to the PSHost object for this interpreter.
    /// </summary>
    private MyHost myHost;

    /// <summary>
    /// Holds a reference to the runspace for this interpreter.
    /// </summary>
    private Runspace myRunSpace;

    /// <summary>
    /// Holds a reference to the currently executing pipeline so that
    /// it can be stopped by the control-C handler.
    /// </summary>
    private PowerShell currentPowerShell;

    /// <summary>
    /// Used to serialize access to instance data.
    /// </summary>
    private object instanceLock = new object();

    /// <summary>
    /// Gets or sets a value indicating whether the host application
    /// should exit.
    /// </summary>
    public bool ShouldExit
    {
        get { return this.shouldExit; }
        set { this.shouldExit = value; }
    }

    /// <summary>
    /// Gets or sets the exit code that the host application will use
    /// when exiting.
    /// </summary>
```

```
/// </summary>
public int ExitCode
{
    get { return this.exitCode; }
    set { this.exitCode = value; }
}

/// <summary>
/// Creates and initiates the listener.
/// </summary>
/// <param name="args">The parameter is not used</param>
private static void Main(string[] args)
{
    // Display the welcome message.
    Console.Title = "Windows PowerShell Console Host Application Sample";
    ConsoleColor oldFg = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine("    Windows PowerShell Console Host Interactive
Sample");
    Console.WriteLine("=====");
    Console.WriteLine(string.Empty);
    Console.WriteLine("This is an example of a simple interactive console
host that uses ");
    Console.WriteLine("the Windows PowerShell engine to interpret
commands.");
    Console.WriteLine("Type 'exit' to exit.");
    Console.WriteLine(string.Empty);
    Console.ForegroundColor = oldFg;

    // Create the listener and run it. This never returns.
    PSListenerConsoleSample listener = new PSListenerConsoleSample();
    listener.Run();
}

/// <summary>
/// Create this instance of the console listener.
/// </summary>
private PSListenerConsoleSample()
{
    // Create the host and runspace instances for this interpreter. Note
that
    // this application doesn't support console files so only the default
snapins
    // will be available.
    this.myHost = new MyHost(this);
    this.myRunSpace = RunspaceFactory.CreateRunspace(this.myHost);
    this.myRunSpace.Open();

    // Create a PowerShell object that will be used to execute the
commands
    // to create $PROFILE and load the profiles.
    lock (this.instanceLock)
    {
        this.currentPowerShell = PowerShell.Create();
```

```

        }

    try
    {
        this.currentPowerShell.Runspace = this.myRunSpace;

        PSCommand[] profileCommands =
Microsoft.Samples.PowerShell.Host.HostUtilities.GetProfileCommands("SampleHo
st04");
        foreach (PSCommand command in profileCommands)
        {
            this.currentPowerShell.Commands = command;
            this.currentPowerShell.Invoke();
        }
    }
    finally
    {
        // Dispose of the pipeline line and set it to null, locked because
currentPowerShell
        // may be accessed by the ctrl-C handler...
        lock (this.instanceLock)
        {
            this.currentPowerShell.Dispose();
            this.currentPowerShell = null;
        }
    }
}

/// <summary>
/// A helper class that builds and executes a pipeline that writes to
the
/// default output path. Any exceptions that are thrown are just passed
to
/// the caller. Since all output goes to the default outputter, this
method
/// returns nothing.
/// </summary>
/// <param name="cmd">The script to run</param>
/// <param name="input">Any input arguments to pass to the script. If
null
/// then nothing is passed in.</param>
private void executeHelper(string cmd, object input)
{
    // Ignore empty command lines.
    if (String.IsNullOrEmpty(cmd))
    {
        return;
    }

    // Create the pipeline object and make it available
    // to the ctrl-C handle through the currentPowerShell instance
    // variable.
    lock (this.instanceLock)
    {
        this.currentPowerShell = PowerShell.Create();
    }
}

```

```
}

this.currentPowerShell.Runspace = this.myRunSpace;

// Create a pipeline for this execution. Place the result in the
// currentPowerShell instance variable so that it is available to
// be stopped.
try
{
    this.currentPowerShell.AddScript(cmd);

    // Now add the default outputter to the end of the pipe and indicate
    // that it should handle both output and errors from the previous
    // commands. This will result in the output being written using the
    PSHost
    // and PSHostUserInterface classes instead of returning objects to
    the hosting
    // application.
    this.currentPowerShell.AddCommand("Out-Default");

this.currentPowerShell.Commands.Commands[0].MergeMyResults(PipelineResultTyp
es.Error, PipelineResultTypes.Output);

    // If there was any input specified, pass it in, otherwise just
    // execute the pipeline.
    if (input != null)
    {
        this.currentPowerShell.Invoke(new object[] { input });
    }
    else
    {
        this.currentPowerShell.Invoke();
    }
}
finally
{
    // Dispose of the pipeline line and set it to null, locked because
    currentPowerShell
    // may be accessed by the ctrl-C handler.
    lock (this.instanceLock)
    {
        this.currentPowerShell.Dispose();
        this.currentPowerShell = null;
    }
}
}

/// <summary>
/// An exception occurred that we want to display
/// using the display formatter. To do this we run
/// a second pipeline passing in the error record.
/// The runtime will bind this to the $input variable
/// which is why $input is being piped to Out-String.
/// We then call WriteErrorLine to make sure the error
/// gets displayed in the correct error color.

```

```

/// </summary>
/// <param name="e">The exception to display</param>
private void ReportException(Exception e)
{
    if (e != null)
    {
        object error;
        IContainsErrorRecord icer = e as IContainsErrorRecord;
        if (icer != null)
        {
            error = icer.ErrorRecord;
        }
        else
        {
            error = (object)new ErrorRecord(e, "Host.ReportException",
ErrorCategory.NotSpecified, null);
        }

        lock (this.instanceLock)
        {
            this.currentPowerShell = PowerShell.Create();
        }

        this.currentPowerShell.Runspace = this.myRunSpace;

        try
        {
            this.currentPowerShell.AddScript("$input").AddCommand("Out-
String");

            // Do not merge errors, this function will swallow errors.
            Collection<PSObject> result;
            PSDataCollection<object> inputCollection = new
            PSDataCollection<object>();
            inputCollection.Add(error);
            inputCollection.Complete();
            result = this.currentPowerShell.Invoke(inputCollection);

            if (result.Count > 0)
            {
                string str = result[0].BaseObject as string;
                if (!string.IsNullOrEmpty(str))
                {
                    // Remove \r\n that is added by Out-String.
                    this.myHost.UI.WriteLine(str.Substring(0, str.Length -
2));
                }
            }
        }
        finally
        {
            // Dispose of the pipeline line and set it to null, locked because
currentPowerShell
            // may be accessed by the ctrl-C handler.
            lock (this.instanceLock)
        }
    }
}

```

```
        {
            this.currentPowerShell.Dispose();
            this.currentPowerShell = null;
        }
    }
}

/// <summary>
/// Basic script execution routine - any runtime exceptions are
/// caught and passed back into the engine to display.
/// </summary>
/// <param name="cmd">The parameter is not used.</param>
private void Execute(string cmd)
{
    try
    {
        // Execute the command with no input.
        this.executeHelper(cmd, null);
    }
    catch (RuntimeException rte)
    {
        this.ReportException(rte);
    }
}

/// <summary>
/// Method used to handle control-C's from the user. It calls the
/// pipeline Stop() method to stop execution. If any exceptions occur
/// they are printed to the console but otherwise ignored.
/// </summary>
/// <param name="sender">See sender property of
ConsoleCancelEventHandler documentation.</param>
/// <param name="e">See e property of ConsoleCancelEventArgs
documentation</param>
private void HandleControlC(object sender, ConsoleCancelEventArgs e)
{
    try
    {
        lock (this.instanceLock)
        {
            if (this.currentPowerShell != null &&
this.currentPowerShell.InvocationStateInfo.State ==
PSInvocationState.Running)
            {
                this.currentPowerShell.Stop();
            }
        }

        e.Cancel = true;
    }
    catch (Exception exception)
    {
        this.myHost.UI.WriteLine(exception.ToString());
    }
}
```

```

    }

    /// <summary>
    /// Implements the basic listener loop. It sets up the ctrl-C handler,
    then
        /// reads a command from the user, executes it and repeats until the
    ShouldExit
        /// flag is set.
    /// </summary>
    private void Run()
    {
        // Set up the control-C handler.
        Console.CancelKeyPress += new
    ConsoleCancelEventHandler(this.HandleControlC);
        Console.TreatControlCAsInput = false;

        // Read commands to execute until ShouldExit is set by
        // the user calling "exit".
        while (!this.ShouldExit)
        {
            this.myHost.UI.WriteLine(ConsoleColor.Cyan, ConsoleColor.Black,
"\nPSConsoleSample: ");
            string cmd = Console.ReadLine();
            this.Execute(cmd);
        }

        // Exit with the desired exit code that was set by exit command.
        // This is set in the host by the MyHost.SetShouldExit()
implementation.
        Environment.Exit(this.ExitCode);
    }
}
}
}

```

Example 2

The following code is the implementation of the [System.Management.Automation.Host.PSHost](#) class that is used by this host application. Those elements that are not implemented throw an exception or return nothing.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Globalization;
    using System.Management.Automation.Host;

    /// <summary>
    /// This is a sample implementation of the PSHost abstract class for
    /// console applications. Not all members are implemented. Those that

```

```
/// are not implemented throw a NotImplementedException exception or
/// return nothing.
/// </summary>
internal class MyHost : PSHost
{
    /// <summary>
    /// A reference to the PSHost implementation.
    /// </summary>
    private PSListenerConsoleSample program;

    /// <summary>
    /// The culture information of the thread that created
    /// this object.
    /// </summary>
    private CultureInfo originalCultureInfo =
        System.Threading.Thread.CurrentThread.CurrentCulture;

    /// <summary>
    /// The UI culture information of the thread that created
    /// this object.
    /// </summary>
    private CultureInfo originalUICultureInfo =
        System.Threading.Thread.CurrentThread.CurrentCulture;

    /// <summary>
    /// The identifier of this PSHost implementation.
    /// </summary>
    private static Guid instanceId = Guid.NewGuid();

    /// <summary>
    /// Initializes a new instance of the MyHost class. Keep
    /// a reference to the host application object so that it
    /// can be informed of when to exit.
    /// </summary>
    /// <param name="program">
    /// A reference to the host application object.
    /// </param>
    public MyHost(PSListenerConsoleSample program)
    {
        this.program = program;
    }

    /// <summary>
    /// A reference to the implementation of the PSHostUserInterface
    /// class for this application.
    /// </summary>
    private MyHostUserInterface myHostUserInterface = new
MyHostUserInterface();

    /// <summary>
    /// Gets the culture information to use. This implementation
    /// returns a snapshot of the culture information of the thread
    /// that created this object.
    /// </summary>
    public override CultureInfo CurrentCulture
```

```
{  
    get { return this.originalCultureInfo; }  
}  
  
/// <summary>  
/// Gets the UI culture information to use. This implementation  
/// returns a snapshot of the UI culture information of the thread  
/// that created this object.  
/// </summary>  
public override CultureInfo CurrentUICulture  
{  
    get { return this.originalUICultureInfo; }  
}  
  
/// <summary>  
/// Gets an identifier for this host. This implementation always  
/// returns the GUID allocated at instantiation time.  
/// </summary>  
public override Guid InstanceId  
{  
    get { return instanceId; }  
}  
  
/// <summary>  
/// Gets a string that contains the name of this host implementation.  
/// Keep in mind that this string may be used by script writers to  
/// identify when your host is being used.  
/// </summary>  
public override string Name  
{  
    get { return "MySampleConsoleHostImplementation"; }  
}  
  
/// <summary>  
/// Gets an instance of the implementation of the PSHostUserInterface  
/// class for this application. This instance is allocated once at  
startup time  
/// and returned every time thereafter.  
/// </summary>  
public override PSHostUserInterface UI  
{  
    get { return this.myHostUserInterface; }  
}  
  
/// <summary>  
/// Gets the version object for this application. Typically this  
/// should match the version resource in the application.  
/// </summary>  
public override Version Version  
{  
    get { return new Version(1, 0, 0, 0); }  
}  
  
/// <summary>  
/// This API Instructs the host to interrupt the currently running
```

```
/// pipeline and start a new nested input loop. In this example this
/// functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
public override void EnterNestedPrompt()
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API instructs the host to exit the currently running input
loop.
/// In this example this functionality is not needed so the method
/// throws a NotImplementedException exception.
/// </summary>
public override void ExitNestedPrompt()
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API is called before an external application process is
/// started. Typically it is used to save state so that the parent
/// can restore state that has been modified by a child process (after
/// the child exits). In this example this functionality is not
/// needed so the method returns nothing.
/// </summary>
public override void NotifyBeginApplication()
{
    return;
}

/// <summary>
/// This API is called after an external application process finishes.
/// Typically it is used to restore state that a child process has
/// altered. In this example, this functionality is not needed so
/// the method returns nothing.
/// </summary>
public override void NotifyEndApplication()
{
    return;
}

/// <summary>
/// Indicate to the host application that exit has
/// been requested. Pass the exit code that the host
/// application should use when exiting the process.
/// </summary>
/// <param name="exitCode">The exit code that the
/// host application should use.</param>
public override void SetShouldExit(int exitCode)
{
    this.program.ShouldExit = true;
```

```
        this.program.ExitCode = exitCode;
    }
}
}
```

Example 3

The following code is the implementation of the [System.Management.Automation.Host.PSHostUserInterface](#) class that is used by this host application.

C#

```
namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Globalization;
    using System.Management.Automation;
    using System.Management.Automation.Host;
    using System.Text;

    /// <summary>
    /// A sample implementation of the PSHostUserInterface abstract class for
    /// console applications. Not all members are implemented. Those that are
    /// not implemented throw a NotImplementedException exception or return
    /// nothing. Members that are implemented include those that map easily to
    /// Console APIs and a basic implementation of the prompt API provided.
    /// </summary>
    internal class MyHostUserInterface : PSHostUserInterface,
    IHostUISupportsMultipleChoiceSelection
    {
        /// <summary>
        /// A reference to the PSRawUserInterface implementation.
        /// </summary>
        private MyRawUserInterface myRawUi = new MyRawUserInterface();

        /// <summary>
        /// Gets an instance of the PSRawUserInterface object for this host
        /// application.
        /// </summary>
        public override PSHostRawUserInterface RawUI
        {
            get { return this.myRawUi; }
        }

        /// <summary>
        /// Prompts the user for input.
        /// <param name="caption">The caption or title of the prompt.</param>
        /// <param name="message">The text of the prompt.</param>
    }
}
```

```

/// <param name="descriptions">A collection of FieldDescription objects
/// that describe each field of the prompt.</param>
/// <returns>A dictionary object that contains the results of the user
/// prompts.</returns>
public override Dictionary<string, PSObject> Prompt(
    string caption,
    string message,
    Collection<FieldDescription> descriptions)
{
    this.WriteLine(
        ConsoleColor.Blue,
        ConsoleColor.Black,
        caption + "\n" + message + " ");
    Dictionary<string, PSObject> results =
        new Dictionary<string, PSObject>();
    foreach (FieldDescription fd in descriptions)
    {
        string[] label = GetHotkeyAndLabel(fd.Label);
        this.WriteLine(label[1]);
        string userData = Console.ReadLine();
        if (userData == null)
        {
            return null;
        }

        results[fd.Name] = PSObject.AsPSObject(userData);
    }

    return results;
}

/// <summary>

/// Provides a set of choices that enable the user to choose a
/// single option from a set of options.
/// </summary>
/// <param name="caption">Text that proceeds (a title) the choices.
</param>
/// <param name="message">A message that describes the choice.</param>
/// <param name="choices">A collection of ChoiceDescription objects that
/// describe each choice.</param>
/// <param name="defaultChoice">The index of the label in the Choices
/// parameter collection. To indicate no default choice, set to -1.
</param>
/// <returns>The index of the Choices parameter collection element that
/// corresponds to the option that is selected by the user.</returns>
public override int PromptForChoice(
    string caption,
    string message,
    Collection<ChoiceDescription>
choices,
    int defaultChoice)
{
    // Write the caption and message strings in Blue.
    this.WriteLine(

```

```

                ConsoleColor.Blue,
                ConsoleColor.Black,
                caption + "\n" + message + "\n");

        // Convert the choice collection into something that is
        // easier to work with. See the BuildHotkeysAndPlainLabels
        // method for details.
        string[,] promptData = BuildHotkeysAndPlainLabels(choices);

        // Format the overall choice prompt string to display.
        StringBuilder sb = new StringBuilder();
        for (int element = 0; element < choices.Count; element++)
        {
            sb.Append(String.Format(
                CultureInfo.CurrentCulture,
                "|{0}> {1} ",
                promptData[0, element],
                promptData[1, element]));
        }

        sb.Append(String.Format(
            CultureInfo.CurrentCulture,
            "[Default is ({0})",
            promptData[0, defaultChoice]));

        // Read prompts until a match is made, the default is
        // chosen, or the loop is interrupted with ctrl-C.
        while (true)
        {
            this.WriteLine(ConsoleColor.Cyan, ConsoleColor.Black,
            sb.ToString());
            string data =
Console.ReadLine().Trim().ToUpper(CultureInfo.CurrentCulture);

            // If the choice string was empty, use the default selection.
            if (data.Length == 0)
            {
                return defaultChoice;
            }

            // See if the selection matched and return the
            // corresponding index if it did.
            for (int i = 0; i < choices.Count; i++)
            {
                if (promptData[0, i] == data)
                {
                    return i;
                }
            }

            this.WriteLine("Invalid choice: " + data);
        }
    }

    #region IHostUISupportsMultipleChoiceSelection Members

```

```

/// <summary>
/// Provides a set of choices that enable the user to choose a one or
/// more options from a set of options.
/// </summary>
/// <param name="caption">Text that proceeds (a title) the choices.
</param>
/// <param name="message">A message that describes the choice.</param>
/// <param name="choices">A collection of ChoiceDescription objects that
/// describe each choice.</param>
/// <param name="defaultChoices">The index of the label in the Choices
/// parameter collection. To indicate no default choice, set to -1.
</param>
/// <returns>The index of the Choices parameter collection element that
/// corresponds to the option that is selected by the user.</returns>
public Collection<int> PromptForChoice(
    string caption,
    string message,
    Collection<ChoiceDescription>
choices,
    IEnumerable<int> defaultChoices)
{
    // Write the caption and message strings in Blue.
    this.WriteLine(
        ConsoleColor.Blue,
        ConsoleColor.Black,
        caption + "\n" + message + "\n");

    // Convert the choice collection into something that is
    // easier to work with. See the BuildHotkeysAndPlainLabels
    // method for details.
    string[,] promptData = BuildHotkeysAndPlainLabels(choices);

    // Format the overall choice prompt string to display.
    StringBuilder sb = new StringBuilder();
    for (int element = 0; element < choices.Count; element++)
    {
        sb.Append(String.Format(
            CultureInfo.CurrentCulture,
            "|{0}> {1} ",
            promptData[0, element],
            promptData[1, element]));
    }

    Collection<int> defaultResults = new Collection<int>();
    if (defaultChoices != null)
    {
        int countDefaults = 0;
        foreach (int defaultChoice in defaultChoices)
        {
            ++countDefaults;
            defaultResults.Add(defaultChoice);
        }

        if (countDefaults != 0)

```

```

    {
        sb.Append(countDefaults == 1 ? "[Default choice is " : "[Default
choices are ");
        foreach (int defaultChoice in defaultChoices)
        {
            sb.AppendFormat(
                CultureInfo.CurrentCulture,
                "\"{0}\",",
                promptData[0, defaultChoice]);
        }

        sb.Remove(sb.Length - 1, 1);
        sb.Append("]");
    }
}

this.WriteLine(
    ConsoleColor.Cyan,
    ConsoleColor.Black,
    sb.ToString());
// Read prompts until a match is made, the default is
// chosen, or the loop is interrupted with ctrl-C.
Collection<int> results = new Collection<int>();
while (true)
{
    ReadNext:
    string prompt = string.Format(CultureInfo.CurrentCulture,
"Choice[{0}]:", results.Count);
    this.Write(ConsoleColor.Cyan, ConsoleColor.Black, prompt);
    string data =
Console.ReadLine().Trim().ToUpper(CultureInfo.CurrentCulture);

    // If the choice string was empty, no more choices have been made.
    // If there were no choices made, return the defaults
    if (data.Length == 0)
    {
        return (results.Count == 0) ? defaultResults : results;
    }

    // See if the selection matched and return the
    // corresponding index if it did.
    for (int i = 0; i < choices.Count; i++)
    {
        if (promptData[0, i] == data)
        {
            results.Add(i);
            goto ReadNext;
        }
    }
}

this.WriteLine("Invalid choice: " + data);
}
}

#endregion

```

```

/// <summary>
/// Prompts the user for credentials with a specified prompt window
/// caption, prompt message, user name, and target name. In this
/// example this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="caption">The caption for the message window.</param>
/// <param name="message">The text of the message.</param>
/// <param name="userName">The user name whose credential is to be
/// prompted for.</param>
/// <param name="targetName">The name of the target for which the
/// credential is collected.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override PSCredential PromptForCredential(
                                         string caption,
                                         string message,
                                         string userName,
                                         string targetName)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// Prompts the user for credentials by using a specified prompt window
/// caption, prompt message, user name and target name, credential
/// types allowed to be returned, and UI behavior options. In this
/// example this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="caption">The caption for the message window.</param>
/// <param name="message">The text of the message.</param>
/// <param name="userName">The user name whose credential is to be
/// prompted for.</param>
/// <param name="targetName">The name of the target for which the
/// credential is collected.</param>
/// <param name="allowedCredentialTypes">A PSCredentialTypes constant
/// that identifies the type of credentials that can be returned.
</param>
/// <param name="options">A PSCredentialUIOptions constant that
/// identifies the UI behavior when it gathers the credentials.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override PSCredential PromptForCredential(
                                         string caption,
                                         string message,
                                         string userName,
                                         string targetName,
                                         PSCredentialTypes
allowedCredentialTypes,
                                         PSCredentialUIOptions options)
{
    throw new NotImplementedException(
        "The method or operation is not
implemented.");
}

```

```
}

/// <summary>
/// Reads characters that are entered by the user until a newline
/// (carriage return) is encountered.
/// </summary>
/// <returns>The characters that are entered by the user.</returns>
public override string ReadLine()
{
    return Console.ReadLine();
}

/// <summary>
/// Reads characters entered by the user until a newline (carriage
return)
/// is encountered and returns the characters as a secure string. In
this
/// example this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <returns>Throws a NotImplementedException exception.</returns>
public override System.Security.SecureString ReadLineAsSecureString()
{
    throw new NotImplementedException("The method or operation is not
implemented.");
}

/// <summary>
/// Writes characters to the output display of the host.
/// </summary>
/// <param name="value">The characters to be written.</param>
public override void Write(string value)
{
    Console.Write(value);
}

/// <summary>
/// Writes characters to the output display of the host with possible
/// foreground and background colors.
/// </summary>
/// <param name="foregroundColor">The color of the characters.</param>
/// <param name="backgroundColor">The background color to use.</param>
/// <param name="value">The characters to be written.</param>
public override void Write(
    ConsoleColor foregroundColor,
    ConsoleColor backgroundColor,
    string value)
{
    ConsoleColor oldFg = Console.ForegroundColor;
    ConsoleColor oldBg = Console.BackgroundColor;
    Console.ForegroundColor = foregroundColor;
    Console.BackgroundColor = backgroundColor;
    Console.Write(value);
    Console.ForegroundColor = oldFg;
    Console.BackgroundColor = oldBg;
```

```
}

/// <summary>
/// Writes a line of characters to the output display of the host
/// with foreground and background colors and appends a newline
/// (carriage return).
/// </summary>
/// <param name="foregroundColor">The foreground color of the display.
</param>
/// <param name="backgroundColor">The background color of the display.
</param>
/// <param name="value">The line to be written.</param>
public override void WriteLine(
    ConsoleColor foregroundColor,
    ConsoleColor backgroundColor,
    string value)
{
    ConsoleColor oldFg = Console.ForegroundColor;
    ConsoleColor oldBg = Console.BackgroundColor;
    Console.ForegroundColor = foregroundColor;
    Console.BackgroundColor = backgroundColor;
    Console.WriteLine(value);
    Console.ForegroundColor = oldFg;
    Console.BackgroundColor = oldBg;
}

/// <summary>
/// Writes a debug message to the output display of the host.
/// </summary>
/// <param name="message">The debug message that is displayed.</param>
public override void WriteDebugLine(string message)
{
    this.WriteLine(
        ConsoleColor.DarkYellow,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "DEBUG: {0}",
message));
}

/// <summary>
/// Writes an error message to the output display of the host.
/// </summary>
/// <param name="value">The error message that is displayed.</param>
public override void WriteErrorLine(string value)
{
    this.WriteLine(
        ConsoleColor.Red,
        ConsoleColor.Black,
        value);
}

/// <summary>
/// Writes a newline character (carriage return)
/// to the output display of the host.
/// </summary>
```

```
public override void WriteLine()
{
    Console.WriteLine();
}

/// <summary>
/// Writes a line of characters to the output display of the host
/// and appends a newline character(carriage return).
/// </summary>
/// <param name="value">The line to be written.</param>
public override void WriteLine(string value)
{
    Console.WriteLine(value);
}

/// <summary>
/// Writes a progress report to the output display of the host.
/// </summary>
/// <param name="sourceId">Unique identifier of the source of the
record. </param>
/// <param name="record">A ProgressReport object.</param>
public override void WriteProgress(long sourceId, ProgressRecord record)
{

}

/// <summary>
/// Writes a verbose message to the output display of the host.
/// </summary>
/// <param name="message">The verbose message that is displayed.</param>
public override void WriteVerboseLine(string message)
{
    this.WriteLine(
        ConsoleColor.Green,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "VERBOSE:
{0}", message));
}

/// <summary>
/// Writes a warning message to the output display of the host.
/// </summary>
/// <param name="message">The warning message that is displayed.</param>
public override void WriteWarningLine(string message)
{
    this.WriteLine(
        ConsoleColor.Yellow,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "WARNING:
{0}", message));
}

/// <summary>
/// Parse a string containing a hotkey character.
/// Take a string of the form
```

```

///> Yes to &all
///> and returns a two-dimensional array split out as
///> "A", "Yes to all".
///</summary>
///<param name="input">The string to process</param>
///<returns>
///> A two dimensional array containing the parsed components.
///</returns>
private static string[] GetHotkeyAndLabel(string input)
{
    string[] result = new string[] { String.Empty, String.Empty };
    string[] fragments = input.Split('&');
    if (fragments.Length == 2)
    {
        if (fragments[1].Length > 0)
        {
            result[0] = fragments[1][0].ToString().
                ToUpper(CultureInfo.CurrentCulture);
        }

        result[1] = (fragments[0] + fragments[1]).Trim();
    }
    else
    {
        result[1] = input;
    }

    return result;
}

///<summary>
///> This is a private worker function splits out the
///> accelerator keys from the menu and builds a two
///> dimensional array with the first access containing the
///> accelerator and the second containing the label string
///> with the & removed.
///</summary>
///<param name="choices">The choice collection to process</param>
///<returns>
///> A two dimensional array containing the accelerator characters
///> and the cleaned-up labels</returns>
private static string[,] BuildHotkeysAndPlainLabels(
    Collection<ChoiceDescription> choices)
{
    // Allocate the result array
    string[,] hotkeysAndPlainLabels = new string[2, choices.Count];

    for (int i = 0; i < choices.Count; ++i)
    {
        string[] hotkeyAndLabel = GetHotkeyAndLabel(choices[i].Label);
        hotkeysAndPlainLabels[0, i] = hotkeyAndLabel[0];
        hotkeysAndPlainLabels[1, i] = hotkeyAndLabel[1];
    }

    return hotkeysAndPlainLabels;
}

```

```
    }
}
}
```

Example 4

The following code is the implementation of the [System.Management.Automation.Host.PSHostRawUserInterface](#) class that is used by this host application. Those elements that are not implemented throw an exception or return nothing.

C#

```
namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Management.Automation.Host;

    /// <summary>
    /// A sample implementation of the PSHostRawUserInterface for console
    /// applications. Members of this class that easily map to the .NET
    /// console class are implemented. More complex methods are not
    /// implemented and throw a NotImplementedException exception.
    /// </summary>
    internal class MyRawUserInterface : PSHostRawUserInterface
    {
        /// <summary>
        /// Gets or sets the background color of text to be written.
        /// This maps to the corresponding Console.BackgroundColor property.
        /// </summary>
        public override ConsoleColor BackgroundColor
        {
            get { return Console.BackgroundColor; }
            set { Console.BackgroundColor = value; }
        }

        /// <summary>
        /// Gets or sets the host buffer size adapted from the Console buffer
        /// size members.
        /// </summary>
        public override Size BufferSize
        {
            get { return new Size(Console.BufferWidth, Console.BufferHeight); }
            set { Console.SetBufferSize(value.Width, value.Height); }
        }

        /// <summary>
        /// Gets or sets the cursor position. In this example this
        /// functionality is not needed so the property throws a
        /// NotImplementedException exception.
        /// </summary>
```

```
public override Coordinates CursorPosition
{
    get { throw new NotImplementedException(
          "The method or operation is not implemented."); }
    set { throw new NotImplementedException(
          "The method or operation is not implemented."); }
}

/// <summary>
/// Gets or sets the cursor size taken directly from the
/// Console.CursorSize property.
/// </summary>
public override int CursorSize
{
    get { return Console.CursorSize; }
    set { Console.CursorSize = value; }
}

/// <summary>
/// Gets or sets the foreground color of the text to be written.
/// This maps to the corresponding Console.ForegroundColor property.
/// </summary>
public override ConsoleColor ForegroundColor
{
    get { return Console.ForegroundColor; }
    set { Console.ForegroundColor = value; }
}

/// <summary>
/// Gets a value indicating whether a key is available. This maps to
/// the corresponding Console.KeyAvailable property.
/// </summary>
public override bool KeyAvailable
{
    get { return Console.KeyAvailable; }
}

/// <summary>
/// Gets the maximum physical size of the window adapted from the
/// Console.LargestWindowWidth and Console.LargestWindowHeight
/// properties.
/// </summary>
public override Size MaxPhysicalWindowSize
{
    get { return new Size(Console.LargestWindowWidth,
Console.LargestWindowHeight); }
}

/// <summary>
/// Gets the maximum window size adapted from the
/// Console.LargestWindowWidth and console.LargestWindowHeight
/// properties.
/// </summary>
public override Size MaxWindowSize
{
```

```
        get { return new Size(Console.LargestScreenWidth,
Console.LargestScreenHeight); }
    }

    /// <summary>
    /// Gets or sets the window position adapted from the Console window
position
    /// members.
    /// </summary>
    public override Coordinates WindowPosition
    {
        get { return new Coordinates(Console.WindowLeft, Console.WindowTop); }
        set { Console.SetWindowPosition(value.X, value.Y); }
    }

    /// <summary>
    /// Gets or sets the window size adapted from the corresponding Console
    /// calls.
    /// </summary>
    public override Size WindowSize
    {
        get { return new Size(Console.WindowWidth, Console.WindowHeight); }
        set { Console.SetWindowSize(value.Width, value.Height); }
    }

    /// <summary>
    /// Gets or sets the title of the window mapped to the Console.Title
    /// property.
    /// </summary>
    public override string WindowTitle
    {
        get { return Console.Title; }
        set { Console.Title = value; }
    }

    /// <summary>
    /// This API resets the input buffer. In this example this
    /// functionality is not needed so the method returns nothing.
    /// </summary>
    public override void FlushInputBuffer()
    {

    }

    /// <summary>
    /// This API returns a rectangular region of the screen buffer. In
    /// this example this functionality is not needed so the method throws
    /// a NotImplementedException exception.
    /// </summary>
    /// <param name="rectangle">Defines the size of the rectangle.</param>
    /// <returns>Throws a NotImplementedException exception.</returns>
    public override BufferCell[,] GetBufferContents(Rectangle rectangle)
    {
        throw new NotImplementedException(
            "The method or operation is not implemented.");
    }
```

```
/// <summary>
/// This API Reads a pressed, released, or pressed and released
keystroke
/// from the keyboard device, blocking processing until a keystroke is
/// typed that matches the specified keystroke options. In this example
/// this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="options">Options, such as IncludeKeyDown, used when
/// reading the keyboard.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override KeyInfo ReadKey(ReadKeyOptions options)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API crops a region of the screen buffer. In this example
/// this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="source">The region of the screen to be scrolled.
</param>
/// <param name="destination">The region of the screen to receive the
/// source region contents.</param>
/// <param name="clip">The region of the screen to include in the
operation.</param>
/// <param name="fill">The character and attributes to be used to fill
all cell.</param>
public override void ScrollBufferContents(Rectangle source, Coordinates
destination, Rectangle clip, BufferCell fill)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API copies an array of buffer cells into the screen buffer
/// at a specified location. In this example this functionality is
/// not needed si the method throws a NotImplementedException
exception.
/// </summary>
/// <param name="origin">The parameter is not used.</param>
/// <param name="contents">The parameter is not used.</param>
public override void SetBufferContents(Coordinates origin, BufferCell[,] 
contents)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API Copies a given character, foreground color, and background
```

```
/// color to a region of the screen buffer. In this example this
/// functionality is not needed so the method throws a
/// NotImplementedException exception./// </summary>
/// <param name="rectangle">Defines the area to be filled. </param>
/// <param name="fill">Defines the fill character.</param>
public override void SetBufferContents(Rectangle rectangle, BufferCell
fill)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}
}
```

See Also

[System.Management.Automation.Host.PSHost](#)

[System.Management.Automation.Host.PSHostUserInterface](#)

[System.Management.Automation.Host.PSHostRawUserInterface](#)

Host05 Sample

Article • 03/24/2025

This sample shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console. This host application also supports calls to remote computers by using the [Enter-PSSession](#) and [Exit-PSSession](#) cmdlets.

Requirements

- This sample requires Windows PowerShell 2.0.
- This application must be run in elevated mode (Run as administrator).

Demonstrates

- Creating a custom host whose classes derive from the [System.Management.Automation.Host.PSHost](#) class, the [System.Management.Automation.Host.PSHostUserInterface](#) class, and the [System.Management.Automation.Host.PSHostRawUserInterface](#) class.
- Building a console application that uses these host classes to build an interactive Windows PowerShell shell.
- Creating a `$PROFILE` variable and loading the following profiles.
 - current user, current host
 - current user, all hosts
 - all users, current host
 - all users, all hosts
- Implement the [System.Management.Automation.Host.IHostUISupportsMultipleChoiceSelection](#) interface.
- Implement the [System.Management.Automation.Host.IHostSupportsInteractiveSession](#) interface to support interactive remoting by using the [Enter-PSSession](#) and [Exit-PSSession](#) cmdlets.

Example 1

This example allows the user to enter commands at a command line, processes those commands, and then prints out the results.

C#

```
namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Host;
    using System.Management.Automation.Runspaces;
    using System.Text;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// Simple PowerShell interactive console host listener implementation.
    This class
        /// implements a basic read-evaluate-print loop or 'listener' allowing you
        to
            /// interactively work with the PowerShell engine.
        /// </summary>
    internal class PSListenerConsoleSample
    {
        /// <summary>
        /// Holds a reference to the runspace for this interpreter.
        /// </summary>
        internal Runspace myRunSpace;

        /// <summary>
        /// Indicator to tell the host application that it should exit.
        /// </summary>
        private bool shouldExit;

        /// <summary>
        /// The exit code that the host application will use to exit.
        /// </summary>
        private int exitCode;

        /// <summary>
        /// Holds a reference to the PSHost object for this interpreter.
        /// </summary>
        private MyHost myHost;

        /// <summary>
        /// Holds a reference to the currently executing pipeline so that
        /// it can be stopped by the control-C handler.
        /// </summary>
        private PowerShell currentPowerShell;

        /// <summary>
        /// Used to serialize access to instance data.
        /// </summary>
    }
}
```

```
private object instanceLock = new object();

/// <summary>
/// Gets or sets a value indicating whether the host application
/// should exit.
/// </summary>
public bool ShouldExit
{
    get { return this.shouldExit; }
    set { this.shouldExit = value; }
}

/// <summary>
/// Gets or sets the exit code that the host application will use
/// when exiting.
/// </summary>
public int ExitCode
{
    get { return this.exitCode; }
    set { this.exitCode = value; }
}

/// <summary>
/// Creates and initiates the listener.
/// </summary>
/// <param name="args">The parameter is not used.</param>
private static void Main(string[] args)
{
    // Display the welcome message.
    Console.Title = "Windows PowerShell Console Host Application Sample";
    ConsoleColor oldFg = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine("    Windows PowerShell Console Host Interactive
Sample");
    Console.WriteLine(
=====
    Console.WriteLine(string.Empty);
    Console.WriteLine("This is an example of a simple interactive console
host that uses ");
    Console.WriteLine("the Windows PowerShell engine to interpret
commands.");
    Console.WriteLine("Type 'exit' to exit.");
    Console.WriteLine(string.Empty);
    Console.ForegroundColor = oldFg;

    // Create the listener and runs it. This method never returns.
    PSLListenerConsoleSample listener = new PSLListenerConsoleSample();
    listener.Run();
}

/// <summary>
/// Create an instance of the console listener.
/// </summary>
private PSLListenerConsoleSample()
{
```

```

// Create the host and runspace instances for this interpreter. Note
// that this application doesn't support console files so only the
// default snap-ins will be available.
this.myHost = new MyHost(this);
this.myRunSpace = RunspaceFactory.CreateRunspace(this.myHost);
this.myRunSpace.Open();

// Create a PowerShell object to run the commands used to create
// $PROFILE and load the profiles.
lock (this.instanceLock)
{
    this.currentPowerShell = PowerShell.Create();
}

try
{
    this.currentPowerShell.Runspace = this.myRunSpace;

    PSCommand[] profileCommands =
Microsoft.Samples.PowerShell.Host.HostUtilities.GetProfileCommands("SampleHo
st05");
    foreach (PSCommand command in profileCommands)
    {
        this.currentPowerShell.Commands = command;
        this.currentPowerShell.Invoke();
    }
}
finally
{
    // Dispose of the pipeline line and set it to null, locked because
currentPowerShell
    // may be accessed by the ctrl-C handler...
    lock (this.instanceLock)
    {
        this.currentPowerShell.Dispose();
        this.currentPowerShell = null;
    }
}
}

/// <summary>
/// A helper class that builds and executes a pipeline that writes to
the
/// default output path. Any exceptions that are thrown are just passed
to
/// the caller. Since all output goes to the default
/// outputter, this method does not return anything.
/// </summary>
/// <param name="cmd">The script to run.</param>
/// <param name="input">Any input arguments to pass to the script.
/// If null then nothing is passed in.</param>
private void executeHelper(string cmd, object input)
{
    // Ignore empty command lines.
    if (String.IsNullOrEmpty(cmd))

```

```
{  
    return;  
}  
  
// Create the pipeline object and make it available to the  
// ctrl-C handle through the currentPowerShell instance  
// variable.  
lock (this.instanceLock)  
{  
    this.currentPowerShell = PowerShell.Create();  
}  
  
// Create a pipeline for this execution, and then place the  
// result in the currentPowerShell variable so it is available  
// to be stopped.  
try  
{  
    this.currentPowerShell.Runspace = this.myRunSpace;  
    this.currentPowerShell.AddScript(cmd);  
  
    // Add the default outputer to the end of the pipe and then  
    // call the MergeMyResults method to merge the output and  
    // error streams from the pipeline. This will result in the  
    // output being written using the PSHost and PSHostUserInterface  
    // classes instead of returning objects to the host application.  
    this.currentPowerShell.AddCommand("Out-Default");  
  
    this.currentPowerShell.Commands.Commands[0].MergeMyResults(PipelineResultTypes.Error, PipelineResultTypes.Output);  
  
    // If there is any input pass it in, otherwise just invoke the  
    // the pipeline.  
    if (input != null)  
    {  
        this.currentPowerShell.Invoke(new object[] { input });  
    }  
    else  
    {  
        this.currentPowerShell.Invoke();  
    }  
}  
finally  
{  
    // Dispose the PowerShell object and set currentPowerShell to null.  
    // It is locked because currentPowerShell may be accessed by the  
    // ctrl-C handler.  
    lock (this.instanceLock)  
    {  
        this.currentPowerShell.Dispose();  
        this.currentPowerShell = null;  
    }  
}  
}  
  
/// <summary>
```

```

/// To display an exception using the display formatter,
/// run a second pipeline passing in the error record.
/// The runtime will bind this to the $input variable,
/// which is why $input is being piped to the Out-String
/// cmdlet. The WriteErrorLine method is called to make sure
/// the error gets displayed in the correct error color.
/// </summary>
/// <param name="e">The exception to display.</param>
private void ReportException(Exception e)
{
    if (e != null)
    {
        object error;
        IContainsErrorRecord icer = e as IContainsErrorRecord;
        if (icer != null)
        {
            error = icer.ErrorRecord;
        }
        else
        {
            error = (object) new ErrorRecord(e, "Host.ReportException",
ErrorCategory.NotSpecified, null);
        }

        lock (this.instanceLock)
        {
            this.currentPowerShell = PowerShell.Create();
        }

        this.currentPowerShell.Runspace = this.myRunSpace;

        try
        {
            this.currentPowerShell.AddScript("$input").AddCommand("Out-
String");

            // Do not merge errors, this function will swallow errors.
            Collection<PSObject> result;
            PSDataCollection<object> inputCollection = new
            PSDataCollection<object>();
            inputCollection.Add(error);
            inputCollection.Complete();
            result = this.currentPowerShell.Invoke(inputCollection);

            if (result.Count > 0)
            {
                string str = result[0].BaseObject as string;
                if (!string.IsNullOrEmpty(str))
                {
                    // Remove \r\n, which is added by the Out-String cmdlet.
                    this.myHost.UI.WriteErrorLine(str.Substring(0, str.Length -
2));
                }
            }
        }
    }
}

```

```
        finally
    {
        // Dispose of the pipeline and set it to null, locking it because
        // currentPowerShell may be accessed by the ctrl-C handler.
        lock (this.instanceLock)
        {
            this.currentPowerShell.Dispose();
            this.currentPowerShell = null;
        }
    }
}

/// <summary>
/// Basic script execution routine. Any runtime exceptions are
/// caught and passed back to the Windows PowerShell engine to
/// display.
/// </summary>
/// <param name="cmd">Script to run.</param>
private void Execute(string cmd)
{
    try
    {
        // Execute the command with no input.
        this.executeHelper(cmd, null);
    }
    catch (RuntimeException rte)
    {
        this.ReportException(rte);
    }
}

/// <summary>
/// Method used to handle control-C's from the user. It calls the
/// pipeline Stop() method to stop execution. If any exceptions occur
/// they are printed to the console but otherwise ignored.
/// </summary>
/// <param name="sender">See sender property documentation of
/// ConsoleCancelEventHandler.</param>
/// <param name="e">See e property documentation of
/// ConsoleCancelEventArgs.</param>
private void HandleControlC(object sender, ConsoleCancelEventArgs e)
{
    try
    {
        lock (this.instanceLock)
        {
            if (this.currentPowerShell != null &&
this.currentPowerShell.InvocationStateInfo.State ==
PSInvocationState.Running)
            {
                this.currentPowerShell.Stop();
            }
        }
    }
}
```

```

        e.Cancel = true;
    }
    catch (Exception exception)
    {
        this.myHost.UI.WriteLine(exception.ToString());
    }
}

/// <summary>
/// Implements the basic listener loop. It sets up the ctrl-C handler,
then
/// reads a command from the user, executes it and repeats until the
ShouldExit
/// flag is set.
/// </summary>
private void Run()
{
    // Set up the control-C handler.
    Console.CancelKeyPress += new
ConsoleCancelEventHandler(this.HandleControlC);
    Console.TreatControlCAsInput = false;

    // Read commands to execute until ShouldExit is set by
    // the user calling "exit".
    while (!this.ShouldExit)
    {
        string prompt;
        if (this.myHost.IsRunspacePushed)
        {
            prompt = string.Format("\n[{0}] PSConsoleSample: ",
this.myRunSpace.ConnectionInfo.ComputerName);
        }
        else
        {
            prompt = "\nPSConsoleSample: ";
        }

        this.myHost.UI.Write(ConsoleColor.Cyan, ConsoleColor.Black, prompt);
        string cmd = Console.ReadLine();
        this.Execute(cmd);
    }

    // Exit with the desired exit code that was set by exit command.
    // This is set in the host by the MyHost.SetShouldExit()
implementation.
    Environment.Exit(this.ExitCode);
}
}
}

```

Example 2

The following code is the implementation of the [System.Management.Automation.Host.PSHost](#) class that is used by this host application. Those elements that are not implemented throw an exception or return nothing.

C#

```
namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Globalization;
    using System.Management.Automation.Host;
    using System.Management.Automation.Runspaces;

    /// <summary>
    /// This is a sample implementation of the PSHost abstract class for
    /// console applications. Not all members are implemented. Those that
    /// are not implemented throw a NotImplementedException exception or
    /// return nothing.
    /// </summary>
    internal class MyHost : PSHost, IHostSupportsInteractiveSession
    {

        /// <summary>
        /// A reference to the PSHost implementation.
        /// </summary>
        private PSListenerConsoleSample program;

        /// <summary>
        /// The culture information of the thread that created
        /// this object.
        /// </summary>
        private CultureInfo originalCultureInfo =
            System.Threading.Thread.CurrentThread.CurrentCulture;

        /// <summary>
        /// The UI culture information of the thread that created
        /// this object.
        /// </summary>
        private CultureInfo originalUICultureInfo =
            System.Threading.Thread.CurrentThread.CurrentUICulture;

        /// <summary>
        /// The identifier of this PSHost implementation.
        /// </summary>
        private static Guid instanceId = Guid.NewGuid();

        /// <summary>
        /// Initializes a new instance of the MyHost class. Keep
        /// a reference to the host application object so that it
        /// can be informed of when to exit.
        /// </summary>
        /// <param name="program">
        /// A reference to the host application object.
    }
}
```

```
/// </param>
public MyHost(PSListenerConsoleSample program)
{
    this.program = program;
}

/// <summary>
/// A reference to the implementation of the PSHostUserInterface
/// class for this application.
/// </summary>
private MyHostUserInterface myHostUserInterface = new
MyHostUserInterface();

/// <summary>
/// A reference to the runspace used to start an interactive session.
/// </summary>
public Runspace pushedRunspace = null;

/// <summary>
/// Gets the culture information to use. This implementation
/// returns a snapshot of the culture information of the thread
/// that created this object.
/// </summary>
public override CultureInfo CurrentCulture
{
    get { return this.originalCultureInfo; }
}

/// <summary>
/// Gets the UI culture information to use. This implementation
/// returns a snapshot of the UI culture information of the thread
/// that created this object.
/// </summary>
public override CultureInfo CurrentUICulture
{
    get { return this.originalUICultureInfo; }
}

/// <summary>
/// Gets an identifier for this host. This implementation always
/// returns the GUID allocated at instantiation time.
/// </summary>
public override Guid InstanceId
{
    get { return instanceId; }
}

/// <summary>
/// Gets a string that contains the name of this host implementation.
/// Keep in mind that this string may be used by script writers to
/// identify when your host is being used.
/// </summary>
public override string Name
{
    get { return "MySampleConsoleHostImplementation"; }
}
```

```
}

/// <summary>
/// Gets an instance of the implementation of the PSHostUserInterface
/// class for this application. This instance is allocated once at
startup time
/// and returned every time thereafter.
/// </summary>
public override PSHostUserInterface UI
{
    get { return this.myHostUserInterface; }
}

/// <summary>
/// Gets the version object for this application. Typically this
/// should match the version resource in the application.
/// </summary>
public override Version Version
{
    get { return new Version(1, 0, 0, 0); }
}

#region IHostSupportsInteractiveSession Properties

/// <summary>
/// Gets a value indicating whether a request
/// to open a PSSession has been made.
/// </summary>
public bool IsRunspacePushed
{
    get { return this.pushedRunspace != null; }
}

/// <summary>
/// Gets or sets the runspace used by the PSSession.
/// </summary>
public Runspace Runspace
{
    get { return this.program.myRunSpace; }
    internal set { this.program.myRunSpace = value; }
}
#endregion IHostSupportsInteractiveSession Properties

/// <summary>
/// This API Instructs the host to interrupt the currently running
/// pipeline and start a new nested input loop. In this example this
/// functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
public override void EnterNestedPrompt()
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}
```

```
/// <summary>
/// This API instructs the host to exit the currently running input
loop.
/// In this example this functionality is not needed so the method
/// throws a NotImplementedException exception.
/// </summary>
public override void ExitNestedPrompt()
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API is called before an external application process is
started. Typically it is used to save state so that the parent
can restore state that has been modified by a child process (after
the child exits). In this example this functionality is not
needed so the method returns nothing.
/// </summary>
public override void NotifyBeginApplication()
{
    return;
}

/// <summary>
/// This API is called after an external application process finishes.
/// Typically it is used to restore state that a child process has
altered. In this example, this functionality is not needed so
the method returns nothing.
/// </summary>
public override void NotifyEndApplication()
{
    return;
}

/// <summary>
/// Indicate to the host application that exit has
been requested. Pass the exit code that the host
application should use when exiting the process.
/// </summary>
/// <param name="exitCode">The exit code that the
host application should use.</param>
public override void SetShouldExit(int exitCode)
{
    this.program.ShouldExit = true;
    this.program.ExitCode = exitCode;
}

#region IHostSupportsInteractiveSession Methods

/// <summary>
/// Requests to close a PSSession.
/// </summary>
public void PopRunspace()
{
```

```

        Runspace = this.pushedRunspace;
        this.pushedRunspace = null;
    }

    /// <summary>
    /// Requests to open a PSSession.
    /// </summary>
    /// <param name="runspace">Runspace to use.</param>
    public void PushRunspace(Runspace runspace)
    {
        this.pushedRunspace = Runspace;
        Runspace = runspace;
    }

    #endregion IHostSupportsInteractiveSession Methods
}
}

```

Example 3

The following code is the implementation of the [System.Management.Automation.Host.PSHostUserInterface](#) class that is used by this host application.

```

C#

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Globalization;
    using System.Management.Automation;
    using System.Management.Automation.Host;
    using System.Text;

    /// <summary>
    /// A sample implementation of the PSHostUserInterface abstract class for
    /// console applications. Not all members are implemented. Those that are
    /// not implemented throw a NotImplementedException exception or return
    /// nothing. Members that are implemented include those that map easily to
    /// Console APIs and a basic implementation of the prompt API provided.
    /// </summary>
    internal class MyHostUserInterface : PSHostUserInterface,
    IHostUISupportsMultipleChoiceSelection
    {
        /// <summary>
        /// A reference to the PSRawUserInterface implementation.
        /// </summary>
        private MyRawUserInterface myRawUi = new MyRawUserInterface();
    }
}

```

```

/// <summary>
/// Gets an instance of the PSRawUserInterface object for this host
/// application.
/// </summary>
public override PSHostRawUserInterface RawUI
{
    get { return this.myRawUi; }
}

/// <summary>
/// Prompts the user for input.
/// <param name="caption">The caption or title of the prompt.</param>
/// <param name="message">The text of the prompt.</param>
/// <param name="descriptions">A collection of FieldDescription objects
/// that describe each field of the prompt.</param>
/// <returns>A dictionary object that contains the results of the user
/// prompts.</returns>
public override Dictionary<string, PSObject> Prompt(
    string caption,
    string message,
    Collection<FieldDescription> descriptions)
{
    this.WriteLine(
        ConsoleColor.Blue,
        ConsoleColor.Black,
        caption + "\n" + message + " ");
    Dictionary<string, PSObject> results =
        new Dictionary<string, PSObject>();
    foreach (FieldDescription fd in descriptions)
    {
        string[] label = GetHotkeyAndLabel(fd.Label);
        this.WriteLine(label[1]);
        string userData = Console.ReadLine();
        if (userData == null)
        {
            return null;
        }

        results[fd.Name] = PSObject.AsPSObject(userData);
    }

    return results;
}

/// <summary>

/// Provides a set of choices that enable the user to choose a
/// single option from a set of options.
/// </summary>
/// <param name="caption">Text that precedes (a title) the choices.
</param>
/// <param name="message">A message that describes the choice.</param>
/// <param name="choices">A collection of ChoiceDescription objects that
/// describe each choice.</param>
/// <param name="defaultChoice">The index of the label in the Choices

```

```

    /// parameter collection. To indicate no default choice, set to -1.
</param>
    /// <returns>The index of the Choices parameter collection element that
    /// corresponds to the option that is selected by the user.</returns>
public override int PromptForChoice(
    string caption,
    string message,
    Collection<ChoiceDescription>
choices,
    int defaultChoice)
{
    // Write the caption and message strings in Blue.
    this.WriteLine(
        ConsoleColor.Blue,
        ConsoleColor.Black,
        caption + "\n" + message + "\n");

    // Convert the choice collection into something that is
    // easier to work with. See the BuildHotkeysAndPlainLabels
    // method for details.
    string[,] promptData = BuildHotkeysAndPlainLabels(choices);

    // Format the overall choice prompt string to display.
    StringBuilder sb = new StringBuilder();
    for (int element = 0; element < choices.Count; element++)
    {
        sb.Append(String.Format(
            CultureInfo.CurrentCulture,
            "|{0}> {1} ",
            promptData[0, element],
            promptData[1, element]));
    }

    sb.Append(String.Format(
        CultureInfo.CurrentCulture,
        "[Default is ({0})",
        promptData[0, defaultChoice]));

    // Read prompts until a match is made, the default is
    // chosen, or the loop is interrupted with ctrl-C.
    while (true)
    {
        this.WriteLine(ConsoleColor.Cyan, ConsoleColor.Black,
sb.ToString());
        string data =
Console.ReadLine().Trim().ToUpper(CultureInfo.CurrentCulture);

        // If the choice string was empty, use the default selection.
        if (data.Length == 0)
        {
            return defaultChoice;
        }

        // See if the selection matched and return the
        // corresponding index if it did.
    }
}

```

```

        for (int i = 0; i < choices.Count; i++)
    {
        if (promptData[0, i] == data)
        {
            return i;
        }
    }

    this.WriteLine("Invalid choice: " + data);
}
}

#region IHostUISupportsMultipleChoiceSelection Members

/// <summary>
/// Provides a set of choices that enable the user to choose one or
/// more options from a set of options.
/// </summary>
/// <param name="caption">Text that precedes (a title) the choices.
</param>
/// <param name="message">A message that describes the choice.</param>
/// <param name="choices">A collection of ChoiceDescription objects that
/// describe each choice.</param>
/// <param name="defaultChoices">The index of the label in the Choices
/// parameter collection. To indicate no default choice, set to -1.
</param>
/// <returns>The index of the Choices parameter collection element that
/// corresponds to the option that is selected by the user.</returns>
public Collection<int> PromptForChoice(
    string caption,
    string message,
    Collection<ChoiceDescription>
choices,
    IEnumerable<int> defaultChoices)
{
    // Write the caption and message strings in Blue.
    this.WriteLine(
        ConsoleColor.Blue,
        ConsoleColor.Black,
        caption + "\n" + message + "\n");

    // Convert the choice collection into something that is
    // easier to work with. See the BuildHotkeysAndPlainLabels
    // method for details.
    string[,] promptData = BuildHotkeysAndPlainLabels(choices);

    // Format the overall choice prompt string to display.
    StringBuilder sb = new StringBuilder();
    for (int element = 0; element < choices.Count; element++)
    {
        sb.Append(String.Format(
            CultureInfo.CurrentCulture,
            "|{0}> {1} ",
            promptData[0, element],
            promptData[1, element]));
    }
}

```

```

}

Collection<int> defaultResults = new Collection<int>();
if (defaultChoices != null)
{
    int countDefaults = 0;
    foreach (int defaultChoice in defaultChoices)
    {
        ++countDefaults;
        defaultResults.Add(defaultChoice);
    }

    if (countDefaults != 0)
    {
        sb.Append(countDefaults == 1 ? "[Default choice is " : "[Default
choices are ");
        foreach (int defaultChoice in defaultChoices)
        {
            sb.AppendFormat(
                CultureInfo.CurrentCulture,
                "\"{0}\",",
                promptData[0, defaultChoice]);
        }

        sb.Remove(sb.Length - 1, 1);
        sb.Append("]");
    }
}

this.WriteLine(
    ConsoleColor.Cyan,
    ConsoleColor.Black,
    sb.ToString());

// Read prompts until a match is made, the default is
// chosen, or the loop is interrupted with ctrl-C.
Collection<int> results = new Collection<int>();
while (true)
{
    ReadNext:
    string prompt = string.Format(CultureInfo.CurrentCulture,
"Choice[{0}]:", results.Count);
    this.Write(ConsoleColor.Cyan, ConsoleColor.Black, prompt);
    string data =
Console.ReadLine().Trim().ToUpper(CultureInfo.CurrentCulture);

    // If the choice string was empty, no more choices have been made.
    // If there were no choices made, return the defaults
    if (data.Length == 0)
    {
        return (results.Count == 0) ? defaultResults : results;
    }

    // See if the selection matched and return the
    // corresponding index if it did.
}

```

```

        for (int i = 0; i < choices.Count; i++)
    {
        if (promptData[0, i] == data)
        {
            results.Add(i);
            goto ReadNext;
        }
    }

    this.WriteLine("Invalid choice: " + data);
}
}

#endregion

/// <summary>
/// Prompts the user for credentials with a specified prompt window
/// caption, prompt message, user name, and target name. In this
/// example this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="caption">The caption for the message window.</param>
/// <param name="message">The text of the message.</param>
/// <param name="userName">The user name whose credential is to be
/// prompted for.</param>
/// <param name="targetName">The name of the target for which the
/// credential is collected.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override PSCredential PromptForCredential(
                                            string caption,
                                            string message,
                                            string userName,
                                            string targetName)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// Prompts the user for credentials by using a specified prompt window
/// caption, prompt message, user name and target name, credential
/// types allowed to be returned, and UI behavior options. In this
/// example this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="caption">The caption for the message window.</param>
/// <param name="message">The text of the message.</param>
/// <param name="userName">The user name whose credential is to be
/// prompted for.</param>
/// <param name="targetName">The name of the target for which the
/// credential is collected.</param>
/// <param name="allowedCredentialTypes">A PSCredentialTypes constant
/// that identifies the type of credentials that can be returned.
</param>
/// <param name="options">A PSCredentialUIOptions constant that

```

```
/// identifies the UI behavior when it gathers the credentials.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override PSCredential PromptForCredential(
    string caption,
    string message,
    string userName,
    string targetName,
    PSCredentialTypes
    allowedCredentialTypes,
    PSCredentialUIOptions options)
{
    throw new NotImplementedException(
        "The method or operation is not
implemented.");
}

/// <summary>
/// Reads characters that are entered by the user until a newline
/// (carriage return) is encountered.
/// </summary>
/// <returns>The characters that are entered by the user.</returns>
public override string ReadLine()
{
    return Console.ReadLine();
}

/// <summary>
/// Reads characters entered by the user until a newline (carriage
return)
/// is encountered and returns the characters as a secure string. In
this
/// example this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <returns>Throws a NotImplementedException exception.</returns>
public override System.Security.SecureString ReadLineAsSecureString()
{
    throw new NotImplementedException("The method or operation is not
implemented.");
}

/// <summary>
/// Writes characters to the output display of the host.
/// </summary>
/// <param name="value">The characters to be written.</param>
public override void Write(string value)
{
    Console.Write(value);
}

/// <summary>
/// Writes characters to the output display of the host with possible
/// foreground and background colors.
/// </summary>
/// <param name="foregroundColor">The color of the characters.</param>
```

```
/// <param name="backgroundColor">The background color to use.</param>
/// <param name="value">The characters to be written.</param>
public override void Write(
    ConsoleColor foregroundColor,
    ConsoleColor backgroundColor,
    string value)
{
    ConsoleColor oldFg = Console.ForegroundColor;
    ConsoleColor oldBg = Console.BackgroundColor;
    Console.ForegroundColor = foregroundColor;
    Console.BackgroundColor = backgroundColor;
    Console.Write(value);
    Console.ForegroundColor = oldFg;
    Console.BackgroundColor = oldBg;
}

/// <summary>
/// Writes a line of characters to the output display of the host
/// with foreground and background colors and appends a newline
/// (carriage return).
/// </summary>
/// <param name="foregroundColor">The foreground color of the display.
</param>
/// <param name="backgroundColor">The background color of the display.
</param>
/// <param name="value">The line to be written.</param>
public override void WriteLine(
    ConsoleColor foregroundColor,
    ConsoleColor backgroundColor,
    string value)
{
    ConsoleColor oldFg = Console.ForegroundColor;
    ConsoleColor oldBg = Console.BackgroundColor;
    Console.ForegroundColor = foregroundColor;
    Console.BackgroundColor = backgroundColor;
    Console.WriteLine(value);
    Console.ForegroundColor = oldFg;
    Console.BackgroundColor = oldBg;
}

/// <summary>
/// Writes a debug message to the output display of the host.
/// </summary>
/// <param name="message">The debug message that is displayed.</param>
public override void WriteDebugLine(string message)
{
    this.WriteLine(
        ConsoleColor.DarkYellow,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "DEBUG: {0}",
message));
}

/// <summary>
/// Writes an error message to the output display of the host.
```

```
/// </summary>
/// <param name="value">The error message that is displayed.</param>
public override void WriteErrorLine(string value)
{
    this.WriteLine(
        ConsoleColor.Red,
        ConsoleColor.Black,
        value);
}

/// <summary>
/// Writes a newline character (carriage return)
/// to the output display of the host.
/// </summary>
public override void WriteLine()
{
    Console.WriteLine();
}

/// <summary>
/// Writes a line of characters to the output display of the host
/// and appends a newline character(carriage return).
/// </summary>
/// <param name="value">The line to be written.</param>
public override void WriteLine(string value)
{
    Console.WriteLine(value);
}

/// <summary>
/// Writes a progress report to the output display of the host.
/// </summary>
/// <param name="sourceId">Unique identifier of the source of the
record. </param>
/// <param name="record">A ProgressReport object.</param>
public override void WriteProgress(long sourceId, ProgressRecord record)
{

}

/// <summary>
/// Writes a verbose message to the output display of the host.
/// </summary>
/// <param name="message">The verbose message that is displayed.</param>
public override void WriteVerboseLine(string message)
{
    this.WriteLine(
        ConsoleColor.Green,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "VERBOSE:
{0}", message));
}

/// <summary>
/// Writes a warning message to the output display of the host.
/// </summary>
```

```
/// </summary>
/// <param name="message">The warning message that is displayed.</param>
public override void WriteWarningLine(string message)
{
    this.WriteLine(
        ConsoleColor.Yellow,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "WARNING:
{0}", message));
}

/// <summary>
/// Parse a string containing a hotkey character.
/// Take a string of the form
///     Yes to &all
/// and returns a two-dimensional array split out as
///     "A", "Yes to all".
/// </summary>
/// <param name="input">The string to process</param>
/// <returns>
/// A two dimensional array containing the parsed components.
/// </returns>
private static string[] GetHotkeyAndLabel(string input)
{
    string[] result = new string[] { String.Empty, String.Empty };
    string[] fragments = input.Split('&');
    if (fragments.Length == 2)
    {
        if (fragments[1].Length > 0)
        {
            result[0] = fragments[1][0].ToString().
                ToUpper(CultureInfo.CurrentCulture);
        }

        result[1] = (fragments[0] + fragments[1]).Trim();
    }
    else
    {
        result[1] = input;
    }

    return result;
}

/// <summary>
/// This is a private worker function splits out the
/// accelerator keys from the menu and builds a two
/// dimensional array with the first access containing the
/// accelerator and the second containing the label string
/// with the & removed.
/// </summary>
/// <param name="choices">The choice collection to process</param>
/// <returns>
/// A two dimensional array containing the accelerator characters
/// and the cleaned-up labels</returns>
```

```

private static string[,] BuildHotkeysAndPlainLabels(
    Collection<ChoiceDescription> choices)
{
    // Allocate the result array
    string[,] hotkeysAndPlainLabels = new string[2, choices.Count];

    for (int i = 0; i < choices.Count; ++i)
    {
        string[] hotkeyAndLabel = GetHotkeyAndLabel(choices[i].Label);
        hotkeysAndPlainLabels[0, i] = hotkeyAndLabel[0];
        hotkeysAndPlainLabels[1, i] = hotkeyAndLabel[1];
    }

    return hotkeysAndPlainLabels;
}
}
}

```

Example 4

The following code is the implementation of the

[System.Management.Automation.Host.PSHostRawUserInterface](#) class that is used by this host application. Those elements that are not implemented throw an exception or return nothing.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Management.Automation.Host;

    /// <summary>
    /// A sample implementation of the PSHostRawUserInterface for console
    /// applications. Members of this class that easily map to the .NET
    /// console class are implemented. More complex methods are not
    /// implemented and throw a NotImplementedException exception.
    /// </summary>
    internal class MyRawUserInterface : PSHostRawUserInterface
    {
        /// <summary>
        /// Gets or sets the background color of text to be written.
        /// This maps to the corresponding Console.BackgroundColor property.
        /// </summary>
        public override ConsoleColor BackgroundColor
        {
            get { return Console.BackgroundColor; }
            set { Console.BackgroundColor = value; }
        }

        /// <summary>

```

```
/// Gets or sets the host buffer size adapted from the Console buffer
/// size members.
/// </summary>
public override Size BufferSize
{
    get { return new Size(Console.BufferWidth, Console.BufferHeight); }
    set { Console.SetBufferSize(value.Width, value.Height); }
}

/// <summary>
/// Gets or sets the cursor position. In this example this
/// functionality is not needed so the property throws a
/// NotImplementedException exception.
/// </summary>
public override Coordinates CursorPosition
{
    get { throw new NotImplementedException(
        "The method or operation is not implemented."); }
    set { throw new NotImplementedException(
        "The method or operation is not implemented."); }
}

/// <summary>
/// Gets or sets the cursor size taken directly from the
/// Console.CursorSize property.
/// </summary>
public override int CursorSize
{
    get { return Console.CursorSize; }
    set { Console.CursorSize = value; }
}

/// <summary>
/// Gets or sets the foreground color of the text to be written.
/// This maps to the corresponding Console.ForegroundColor property.
/// </summary>
public override ConsoleColor ForegroundColor
{
    get { return Console.ForegroundColor; }
    set { Console.ForegroundColor = value; }
}

/// <summary>
/// Gets a value indicating whether a key is available. This maps to
/// the corresponding Console.KeyAvailable property.
/// </summary>
public override bool KeyAvailable
{
    get { return Console.KeyAvailable; }
}

/// <summary>
/// Gets the maximum physical size of the window adapted from the
/// Console.LargestWindowWidth and Console.LargestWindowHeight
/// properties.

```

```
/// </summary>
public override Size MaxPhysicalWindowSize
{
    get { return new Size(Console.LargestScreenWidth,
Console.LargestScreenHeight); }
}

/// <summary>
/// Gets the maximum window size adapted from the
/// Console.LargestScreenWidth and console.LargestScreenHeight
/// properties.
/// </summary>
public override Size MaxWindowSize
{
    get { return new Size(Console.LargestScreenWidth,
Console.LargestScreenHeight); }
}

/// <summary>
/// Gets or sets the window position adapted from the Console window
position
/// members.
/// </summary>
public override Coordinates WindowPosition
{
    get { return new Coordinates(Console.WindowLeft, Console.WindowTop); }
    set { Console.SetWindowPosition(value.X, value.Y); }
}

/// <summary>
/// Gets or sets the window size adapted from the corresponding Console
/// calls.
/// </summary>
public override Size WindowSize
{
    get { return new Size(Console.WindowWidth, Console.WindowHeight); }
    set { Console.SetWindowSize(value.Width, value.Height); }
}

/// <summary>
/// Gets or sets the title of the window mapped to the Console.Title
/// property.
/// </summary>
public override string WindowTitle
{
    get { return Console.Title; }
    set { Console.Title = value; }
}

/// <summary>
/// This API resets the input buffer. In this example this
/// functionality is not needed so the method returns nothing.
/// </summary>
public override void FlushInputBuffer()
{
```

```
}

/// <summary>
/// This API returns a rectangular region of the screen buffer. In
/// this example this functionality is not needed so the method throws
/// a NotImplementedException exception.
/// </summary>
/// <param name="rectangle">Defines the size of the rectangle.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override BufferCell[,] GetBufferContents(Rectangle rectangle)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API Reads a pressed, released, or pressed and released
keystroke
/// from the keyboard device, blocking processing until a keystroke is
/// typed that matches the specified keystroke options. In this example
/// this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="options">Options, such as IncludeKeyDown, used when
/// reading the keyboard.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override KeyInfo ReadKey(ReadKeyOptions options)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API crops a region of the screen buffer. In this example
/// this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="source">The region of the screen to be scrolled.
</param>
/// <param name="destination">The region of the screen to receive the
/// source region contents.</param>
/// <param name="clip">The region of the screen to include in the
operation.</param>
/// <param name="fill">The character and attributes to be used to fill
all cell.</param>
public override void ScrollBufferContents(Rectangle source, Coordinates
destination, Rectangle clip, BufferCell fill)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API copies an array of buffer cells into the screen buffer
/// at a specified location. In this example this functionality is
```

```
    /// not needed si the method throws a NotImplementedException  
    exception.  
    /// </summary>  
    /// <param name="origin">The parameter is not used.</param>  
    /// <param name="contents">The parameter is not used.</param>  
    public override void SetBufferContents(Coordinates origin, BufferCell[,]  
contents)  
{  
    throw new NotImplementedException(  
        "The method or operation is not implemented.");  
}  
  
    /// <summary>  
    /// This API Copies a given character, foreground color, and background  
    /// color to a region of the screen buffer. In this example this  
    /// functionality is not needed so the method throws a  
    /// NotImplementedException exception./// </summary>  
    /// <param name="rectangle">Defines the area to be filled. </param>  
    /// <param name="fill">Defines the fill character.</param>  
    public override void SetBufferContents(Rectangle rectangle, BufferCell  
fill)  
{  
    throw new NotImplementedException(  
        "The method or operation is not implemented.");  
}  
}
```

See Also

[System.Management.Automation.Host.PSHost](#)

[System.Management.Automation.Host.PSHostUserInterface](#)

[System.Management.Automation.Host.PSHostRawUserInterface](#)

Host06 Sample

Article • 03/24/2025

This sample shows how to build an interactive console-based host application that reads commands from the command line, executes the commands, and then displays the results to the console. In addition, this sample uses the Tokenizer APIs to specify the color of the text that is entered by the user.

Requirements

- This sample requires Windows PowerShell 2.0.
- This application must be run in elevated mode (Run as administrator).

Demonstrates

- Creating a custom host whose classes derive from the [System.Management.Automation.Host.PSHost](#) class, the [System.Management.Automation.Host.PSHostUserInterface](#) class, and the [System.Management.Automation.Host.PSHostRawUserInterface](#) class.
- Building a console application that uses these host classes to build an interactive Windows PowerShell shell.
- Creating a `$PROFILE` variable and loading the following profiles.
 - current user, current host
 - current user, all hosts
 - all users, current host
 - all users, all hosts
- Implement the [System.Management.Automation.Host.IHostUISupportsMultipleChoiceSelection](#) interface.
- Implement the [System.Management.Automation.Host.IHostSupportsInteractiveSession](#) interface to support interactive remoting by using the [Enter-PSSession](#) and [Exit-PSSession](#) cmdlets.
- Use the Tokenize API to colorize the command line as it is typed.

Example 1

This example allows the user to enter commands at a command line, processes those commands, and then prints out the results.

C#

```
namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Host;
    using System.Management.Automation.Runspaces;
    using System.Text;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This sample shows how to implement a basic read-evaluate-print
    /// loop (or 'listener') that allowing you to interactively work
    /// with the Windows PowerShell engine.
    /// </summary>
    internal class PSListenerConsoleSample
    {
        /// <summary>
        /// Used to read user input.
        /// </summary>
        internal ConsoleReadLine consoleReadLine = new ConsoleReadLine();

        /// <summary>
        /// Holds a reference to the runspace for this interpreter.
        /// </summary>
        internal Runspace myRunSpace;

        /// <summary>
        /// Indicator to tell the host application that it should exit.
        /// </summary>
        private bool shouldExit;

        /// <summary>
        /// The exit code that the host application will use to exit.
        /// </summary>
        private int exitCode;

        /// <summary>
        /// Holds a reference to the PSHost implementation for this interpreter.
        /// </summary>
        private MyHost myHost;

        /// <summary>
        /// Holds a reference to the currently executing pipeline so that it can
        be
```

```
/// stopped by the control-C handler.
/// </summary>
private PowerShell currentPowerShell;

/// <summary>
/// Used to serialize access to instance data.
/// </summary>
private object instanceLock = new object();

/// <summary>
/// Gets or sets a value indicating whether the host application
/// should exit.
/// </summary>
public bool ShouldExit
{
    get { return this.shouldExit; }
    set { this.shouldExit = value; }
}

/// <summary>
/// Gets or sets a value indicating whether the host application
/// should exit.
/// </summary>
public int ExitCode
{
    get { return this.exitCode; }
    set { this.exitCode = value; }
}

/// <summary>
/// Creates and initiates the listener instance.
/// </summary>
/// <param name="args">This parameter is not used.</param>
private static void Main(string[] args)
{
    // Display the welcome message.
    Console.Title = "PowerShell Console Host Sample Application";
    ConsoleColor oldFg = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.WriteLine("    Windows PowerShell Console Host Application
Sample");
    Console.WriteLine("=====");
    Console.WriteLine(string.Empty);
    Console.WriteLine("This is an example of a simple interactive console
host uses ");
    Console.WriteLine("the Windows PowerShell engine to interpret
commands.");
    Console.WriteLine("Type 'exit' to exit.");
    Console.WriteLine(string.Empty);
    Console.ForegroundColor = oldFg;

    // Create the listener and run it. This method never returns.
    PSLListenerConsoleSample listener = new PSLListenerConsoleSample();
    listener.Run();
}
```

```
}

/// <summary>
/// Initializes a new instance of the PSListenerConsoleSample class.
/// </summary>
public PSListenerConsoleSample()
{
    // Create the host and runspace instances for this interpreter.
    // Note that this application does not support console files so
    // only the default snap-ins will be available.
    this.myHost = new MyHost(this);
    this.myRunSpace = RunspaceFactory.CreateRunspace(this.myHost);
    this.myRunSpace.Open();

    // Create a PowerShell object to run the commands used to create
    // $PROFILE and load the profiles.
    lock (this.instanceLock)
    {
        this.currentPowerShell = PowerShell.Create();
    }

    try
    {
        this.currentPowerShell.Runspace = this.myRunSpace;

        PSCommand[] profileCommands =
Microsoft.Samples.PowerShell.Host.HostUtilities.GetProfileCommands("SampleHo
st06");
        foreach (PSCommand command in profileCommands)
        {
            this.currentPowerShell.Commands = command;
            this.currentPowerShell.Invoke();
        }
    }
    finally
    {
        // Dispose the PowerShell object and set currentPowerShell
        // to null. It is locked because currentPowerShell may be
        // accessed by the ctrl-C handler.
        lock (this.instanceLock)
        {
            this.currentPowerShell.Dispose();
            this.currentPowerShell = null;
        }
    }
}

/// <summary>
/// A helper class that builds and executes a pipeline that writes
/// to the default output path. Any exceptions that are thrown are
/// just passed to the caller. Since all output goes to the default
/// outputter, this method does not return anything.
/// </summary>
/// <param name="cmd">The script to run.</param>
/// <param name="input">Any input arguments to pass to the script.
```

```

/// If null then nothing is passed in.</param>
private void executeHelper(string cmd, object input)
{
    // Ignore empty command lines.
    if (String.IsNullOrEmpty(cmd))
    {
        return;
    }

    // Create the pipeline object and make it available to the
    // ctrl-C handle through the currentPowerShell instance
    // variable.
    lock (this.instanceLock)
    {
        this.currentPowerShell = PowerShell.Create();
    }

    // Add a script and command to the pipeline and then run the pipeline.
Place
    // the results in the currentPowerShell variable so that the pipeline
can be
    // stopped.
    try
    {
        this.currentPowerShell.Runspace = this.myRunSpace;

        this.currentPowerShell.AddScript(cmd);

        // Add the default outputter to the end of the pipe and then call
the
        // MergeMyResults method to merge the output and error streams from
the
        // pipeline. This will result in the output being written using the
PShell
        // and PShellUserInterface classes instead of returning objects to
the host
        // application.
        this.currentPowerShell.AddCommand("Out-Default");

this.currentPowerShell.Commands.Commands[0].MergeMyResults(PipelineResultTyp
es.Error, PipelineResultTypes.Output);

        // If there is any input pass it in, otherwise just invoke the
        // the pipeline.
        if (input != null)
        {
            this.currentPowerShell.Invoke(new object[] { input });
        }
        else
        {
            this.currentPowerShell.Invoke();
        }
    }
    finally
    {

```

```

        // Dispose the PowerShell object and set currentPowerShell to null.
        // It is locked because currentPowerShell may be accessed by the
        // ctrl-C handler.
        lock (this.instanceLock)
        {
            this.currentPowerShell.Dispose();
            this.currentPowerShell = null;
        }
    }

    /// <summary>
    /// To display an exception using the display formatter,
    /// run a second pipeline passing in the error record.
    /// The runtime will bind this to the $input variable,
    /// which is why $input is being piped to the Out-String
    /// cmdlet. The WriteErrorLine method is called to make sure
    /// the error gets displayed in the correct error color.
    /// </summary>
    /// <param name="e">The exception to display.</param>
    private void ReportException(Exception e)
    {
        if (e != null)
        {
            object error;
            IContainsErrorRecord icer = e as IContainsErrorRecord;
            if (icer != null)
            {
                error = icer.ErrorRecord;
            }
            else
            {
                error = (object)new ErrorRecord(e, "Host.ReportException",
                    ErrorCategory.NotSpecified, null);
            }

            lock (this.instanceLock)
            {
                this.currentPowerShell = PowerShell.Create();
            }

            this.currentPowerShell.Runspace = this.myRunSpace;

            try
            {
                this.currentPowerShell.AddScript("$input").AddCommand("Out-
String");

                // Do not merge errors, this function will swallow errors.
                Collection<PSObject> result;
                PSDataCollection<object> inputCollection = new
                PSDataCollection<object>();
                inputCollection.Add(error);
                inputCollection.Complete();
                result = this.currentPowerShell.Invoke(inputCollection);
            }
        }
    }
}

```

```

        if (result.Count > 0)
        {
            string str = result[0].BaseObject as string;
            if (!string.IsNullOrEmpty(str))
            {
                // Remove \r\n, which is added by the Out-String cmdlet.
                this.myHost.UI.WriteLine(str.Substring(0, str.Length -
2));
            }
        }
    }
    finally
    {
        // Dispose of the pipeline and set it to null, locking it because
        // currentPowerShell may be accessed by the ctrl-C handler.
        lock (this.instanceLock)
        {
            this.currentPowerShell.Dispose();
            this.currentPowerShell = null;
        }
    }
}

/// <summary>
/// Basic script execution routine. Any runtime exceptions are
/// caught and passed back to the Windows PowerShell engine to
/// display.
/// </summary>
/// <param name="cmd">Script to run.</param>
private void Execute(string cmd)
{
    try
    {
        // Run the command with no input.
        this.executeHelper(cmd, null);
    }
    catch (RuntimeException rte)
    {
        this.ReportException(rte);
    }
}

/// <summary>
/// Method used to handle control-C's from the user. It calls the
/// pipeline Stop() method to stop execution. If any exceptions occur
/// they are printed to the console but otherwise ignored.
/// </summary>
/// <param name="sender">See sender property documentation of
/// ConsoleCancelEventHandler.</param>
/// <param name="e">See e property documentation of
/// ConsoleCancelEventArgs.</param>
private void HandleControlC(object sender, ConsoleCancelEventArgs e)
{

```

```

try
{
    lock (this.instanceLock)
    {
        if (this.currentPowerShell != null &&
this.currentPowerShell.InvocationStateInfo.State ==
PSInvocationState.Running)
        {
            this.currentPowerShell.Stop();
        }
    }

    e.Cancel = true;
}
catch (Exception exception)
{
    this.myHost.UI.WriteLine(exception.ToString());
}
}

/// <summary>
/// Implements the basic listener loop. It sets up the ctrl-C handler,
then
/// reads a command from the user, executes it and repeats until the
ShouldExit
/// flag is set.
/// </summary>
private void Run()
{
    // Set up the control-C handler.
    Console.CancelKeyPress += new
ConsoleCancelEventHandler(this.HandleControlC);
    Console.TreatControlCAsInput = false;

    // Read commands and run them until the ShouldExit flag is set by
    // the user calling "exit".
    while (!this.ShouldExit)
    {
        string prompt;
        if (this.myHost.IsRunspacePushed)
        {
            prompt = string.Format("\n[{0}] PSConsoleSample: ",
this.myRunSpace.ConnectionInfo.ComputerName);
        }
        else
        {
            prompt = "\nPSConsoleSample: ";
        }

        this.myHost.UI.WriteLine(ConsoleColor.Cyan, ConsoleColor.Black, prompt);
        string cmd = this.consoleReadLine.ReadLine();
        this.Execute(cmd);
    }

    // Exit with the desired exit code that was set by the exit command.
}

```

```

        // The exit code is set in the host by the MyHost.SetShouldExit()
        method.
        Environment.Exit(this.ExitCode);
    }
}
}

```

Example 2

The following code is the implementation of the [System.Management.Automation.Host.PSHost](#) class that is used by this host application. Those elements that are not implemented throw an exception or return nothing.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Globalization;
    using System.Management.Automation.Host;
    using System.Management.Automation.Runspaces;

    /// <summary>
    /// This is a sample implementation of the PSHost abstract class for
    /// console applications. Not all members are implemented. Those that
    /// are not implemented throw a NotImplementedException exception or
    /// return nothing.
    /// </summary>
    internal class MyHost : PSHost, IHostSupportsInteractiveSession
    {
        public MyHost(PSListenerConsoleSample program)
        {
            this.program = program;
        }

        /// <summary>
        /// A reference to the PSHost implementation.
        /// </summary>
        private PSListenerConsoleSample program;

        /// <summary>
        /// The culture information of the thread that created
        /// this object.
        /// </summary>
        private CultureInfo originalCultureInfo =
            System.Threading.Thread.CurrentThread.CurrentCulture;

        /// <summary>
        /// The UI culture information of the thread that created
        /// this object.
        /// </summary>
    }
}

```

```
private CultureInfo originalUICultureInfo =
    System.Threading.Thread.CurrentThread.CurrentCulture;

/// <summary>
/// The identifier of this PSHost implementation.
/// </summary>
private static Guid instanceId = Guid.NewGuid();

/// <summary>
/// A reference to the implementation of the PSHostUserInterface
/// class for this application.
/// </summary>
private MyHostUserInterface myHostUserInterface = new
MyHostUserInterface();

/// <summary>
/// A reference to the runspace used to start an interactive session.
/// </summary>
public Runspace pushedRunspace = null;

/// <summary>
/// Gets the culture information to use. This implementation
/// returns a snapshot of the culture information of the thread
/// that created this object.
/// </summary>
public override CultureInfo CurrentCulture
{
    get { return this.originalCultureInfo; }
}

/// <summary>
/// Gets the UI culture information to use. This implementation
/// returns a snapshot of the UI culture information of the thread
/// that created this object.
/// </summary>
public override CultureInfo CurrentUICulture
{
    get { return this.originalUICultureInfo; }
}

/// <summary>
/// Gets an identifier for this host. This implementation always
/// returns the GUID allocated at instantiation time.
/// </summary>
public override Guid InstanceId
{
    get { return instanceId; }
}

/// <summary>
/// Gets a string that contains the name of this host implementation.
/// Keep in mind that this string may be used by script writers to
/// identify when your host is being used.
/// </summary>
public override string Name
```

```

{
    get { return "MySampleConsoleHostImplementation"; }
}

/// <summary>
/// Gets an instance of the implementation of the PSHostUserInterface
/// class for this application. This instance is allocated once at
startup time
/// and returned every time thereafter.
/// </summary>
public override PSHostUserInterface UI
{
    get { return this.myHostUserInterface; }
}

/// <summary>
/// Gets the version object for this application. Typically this
/// should match the version resource in the application.
/// </summary>
public override Version Version
{
    get { return new Version(1, 0, 0, 0); }
}

#region IHostSupportsInteractiveSession Properties

/// <summary>
/// Gets a value indicating whether a request
/// to open a PSSession has been made.
/// </summary>
public bool IsRunspacePushed
{
    get { return this.pushedRunspace != null; }
}

/// <summary>
/// Gets or sets the runspace used by the PSSession.
/// </summary>
public Runspace Runspace
{
    get { return this.program.myRunSpace; }
    internal set { this.program.myRunSpace = value; }
}
#endregion IHostSupportsInteractiveSession Properties

/// <summary>
/// This API Instructs the host to interrupt the currently running
/// pipeline and start a new nested input loop. In this example this
/// functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
public override void EnterNestedPrompt()
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

```

```
}

/// <summary>
/// This API instructs the host to exit the currently running input
loop.
/// In this example this functionality is not needed so the method
/// throws a NotImplementedException exception.
/// </summary>
public override void ExitNestedPrompt()
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// This API is called before an external application process is
started. Typically it is used to save state so that the parent
can restore state that has been modified by a child process (after
the child exits). In this example this functionality is not
needed so the method returns nothing.
/// </summary>
public override void NotifyBeginApplication()
{
    return;
}

/// <summary>
/// This API is called after an external application process finishes.
/// Typically it is used to restore state that a child process has
altered. In this example, this functionality is not needed so
the method returns nothing.
/// </summary>
public override void NotifyEndApplication()
{
    return;
}

/// <summary>
/// Indicate to the host application that exit has
been requested. Pass the exit code that the host
application should use when exiting the process.
/// </summary>
/// <param name="exitCode">The exit code that the
/// host application should use.</param>
public override void SetShouldExit(int exitCode)
{
    this.program.ShouldExit = true;
    this.program.ExitCode = exitCode;
}

#region IHostSupportsInteractiveSession Methods

/// <summary>
/// Requests to close a PSSession.
/// </summary>
```

```

    public void PopRunspace()
    {
        Runspace = this.pushedRunspace;
        this.pushedRunspace = null;
    }

    /// <summary>
    /// Requests to open a PSSession.
    /// </summary>
    /// <param name="runspace">Runspace to use.</param>
    public void PushRunspace(Runspace runspace)
    {
        this.pushedRunspace = Runspace;
        Runspace = runspace;
    }

    #endregion IHostSupportsInteractiveSession Methods
}
}

```

Example 3

The following code is the implementation of the [System.Management.Automation.Host.PSHostUserInterface](#) class that is used by this host application.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Globalization;
    using System.Management.Automation;
    using System.Management.Automation.Host;
    using System.Text;

    /// <summary>
    /// A sample implementation of the PSHostUserInterface abstract class for
    /// console applications. Not all members are implemented. Those that are
    /// not implemented throw a NotImplementedException exception or return
    /// nothing. Members that are implemented include those that map easily to
    /// Console APIs and a basic implementation of the prompt API provided.
    /// </summary>
    internal class MyHostUserInterface : PSHostUserInterface,
    IHostUISupportsMultipleChoiceSelection
    {
        /// <summary>
        /// A reference to the PSRawUserInterface implementation.
        /// </summary>

```

```

private MyRawUserInterface myRawUi = new MyRawUserInterface();

/// <summary>
/// Gets an instance of the PSRawUserInterface class for this host
/// application.
/// </summary>
public override PSHostRawUserInterface RawUI
{
    get { return this.myRawUi; }
}

/// <summary>
/// Prompts the user for input.
/// <param name="caption">The caption or title of the prompt.</param>
/// <param name="message">The text of the prompt.</param>
/// <param name="descriptions">A collection of FieldDescription objects
/// that describe each field of the prompt.</param>
/// <returns>A dictionary object that contains the results of the user
/// prompts.</returns>
public override Dictionary<string, PSObject> Prompt(
    string caption,
    string message,
    Collection<FieldDescription> descriptions)
{
    this.WriteLine(
        ConsoleColor.DarkCyan,
        ConsoleColor.Black,
        caption + "\n" + message + " ");
    Dictionary<string, PSObject> results =
        new Dictionary<string, PSObject>();
    foreach (FieldDescription fd in descriptions)
    {
        string[] label = GetHotkeyAndLabel(fd.Label);
        this.WriteLine(label[1]);
        string userData = Console.ReadLine();
        if (userData == null)
        {
            return null;
        }

        results[fd.Name] = PSObject.AsPSObject(userData);
    }

    return results;
}

/// <summary>

/// Provides a set of choices that enable the user to choose a
/// single option from a set of options.
/// </summary>
/// <param name="caption">Text that precedes (a title) the choices.
</param>
/// <param name="message">A message that describes the choice.</param>
/// <param name="choices">A collection of ChoiceDescription objects that

```

```

/// describe each choice.</param>
/// <param name="defaultChoice">The index of the label in the Choices
/// parameter collection. To indicate no default choice, set to -1.
</param>
/// <returns>The index of the Choices parameter collection element that
/// corresponds to the option that is selected by the user.</returns>
public override int PromptForChoice(
    string caption,
    string message,
    Collection<ChoiceDescription>
choices,
    int defaultChoice)
{
    // Write the caption and message strings in Blue.
    this.WriteLine(
        ConsoleColor.Blue,
        ConsoleColor.Black,
        caption + "\n" + message + "\n");

    // Convert the choice collection into something that is
    // easier to work with. See the BuildHotkeysAndPlainLabels
    // method for details.
    string[,] promptData = BuildHotkeysAndPlainLabels(choices);

    // Format the overall choice prompt string to display.
    StringBuilder sb = new StringBuilder();
    for (int element = 0; element < choices.Count; element++)
    {
        sb.Append(String.Format(
            CultureInfo.CurrentCulture,
            "|{0}> {1} ",
            promptData[0, element],
            promptData[1, element]));
    }

    sb.Append(String.Format(
        CultureInfo.CurrentCulture,
        "[Default is ({0})",
        promptData[0, defaultChoice]));

    // Read prompts until a match is made, the default is
    // chosen, or the loop is interrupted with ctrl-C.
    while (true)
    {
        this.WriteLine(ConsoleColor.Cyan, ConsoleColor.Black,
sb.ToString());
        string data =
Console.ReadLine().Trim().ToUpper(CultureInfo.CurrentCulture);

        // If the choice string was empty, use the default selection.
        if (data.Length == 0)
        {
            return defaultChoice;
        }
    }
}

```

```

        // See if the selection matched and return the
        // corresponding index if it did.
        for (int i = 0; i < choices.Count; i++)
        {
            if (promptData[0, i] == data)
            {
                return i;
            }
        }

        this.WriteLine("Invalid choice: " + data);
    }
}

#region IHostUISupportsMultipleChoiceSelection Members

    /// <summary>
    /// Provides a set of choices that enable the user to choose a one or
    /// more options from a set of options.
    /// </summary>
    /// <param name="caption">Text that proceeds (a title) the choices.
</param>
    /// <param name="message">A message that describes the choice.</param>
    /// <param name="choices">A collection of ChoiceDescription objects that
    /// describe each choice.</param>
    /// <param name="defaultChoices">The index of the label in the Choices
    /// parameter collection. To indicate no default choice, set to -1.
</param>
    /// <returns>The index of the Choices parameter collection element that
    /// corresponds to the option that is selected by the user.</returns>
public Collection<int> PromptForChoice(
    string caption,
    string message,
    Collection<ChoiceDescription>
choices,
    IEnumerable<int> defaultChoices)
{
    // Write the caption and message strings in Blue.
    this.WriteLine(
        ConsoleColor.Blue,
        ConsoleColor.Black,
        caption + "\n" + message + "\n");

    // Convert the choice collection into something that is
    // easier to work with. See the BuildHotkeysAndPlainLabels
    // method for details.
    string[,] promptData = BuildHotkeysAndPlainLabels(choices);

    // Format the overall choice prompt string to display.
    StringBuilder sb = new StringBuilder();
    for (int element = 0; element < choices.Count; element++)
    {
        sb.Append(String.Format(
            CultureInfo.CurrentCulture,
            "|{0}> {1} ",

```

```

                promptData[0, element],
                promptData[1, element]));
}

Collection<int> defaultResults = new Collection<int>();
if (defaultChoices != null)
{
    int countDefaults = 0;
    foreach (int defaultChoice in defaultChoices)
    {
        ++countDefaults;
        defaultResults.Add(defaultChoice);
    }

    if (countDefaults != 0)
    {
        sb.Append(countDefaults == 1 ? "[Default choice is " : "[Default
choices are ");
        foreach (int defaultChoice in defaultChoices)
        {
            sb.AppendFormat(
                CultureInfo.CurrentCulture,
                "\'{0}\',",
                promptData[0, defaultChoice]);
        }

        sb.Remove(sb.Length - 1, 1);
        sb.Append("]");
    }
}

this.WriteLine(
    ConsoleColor.Cyan,
    ConsoleColor.Black,
    sb.ToString());
// Read prompts until a match is made, the default is
// chosen, or the loop is interrupted with ctrl-C.
Collection<int> results = new Collection<int>();
while (true)
{
    ReadNext:
    string prompt = string.Format(CultureInfo.CurrentCulture,
"Choice[{0}]:", results.Count);
    this.Write(ConsoleColor.Cyan, ConsoleColor.Black, prompt);
    string data =
Console.ReadLine().Trim().ToUpper(CultureInfo.CurrentCulture);

    // If the choice string was empty, no more choices have been made.
    // If there were no choices made, return the defaults
    if (data.Length == 0)
    {
        return (results.Count == 0) ? defaultResults : results;
    }

    // See if the selection matched and return the
}

```

```

        // corresponding index if it did.
        for (int i = 0; i < choices.Count; i++)
        {
            if (promptData[0, i] == data)
            {
                results.Add(i);
                goto ReadNext;
            }
        }

        this.WriteLine("Invalid choice: " + data);
    }
}

#endregion

/// <summary>
/// Prompts the user for credentials with a specified prompt window
/// caption, prompt message, user name, and target name. In this
/// example this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="caption">The caption for the message window.</param>
/// <param name="message">The text of the message.</param>
/// <param name="userName">The user name whose credential is to be
/// prompted for.</param>
/// <param name="targetName">The name of the target for which the
/// credential is collected.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override PSCredential PromptForCredential(
    string caption,
    string message,
    string userName,
    string targetName)
{
    throw new NotImplementedException(
        "The method or operation is not implemented.");
}

/// <summary>
/// Prompts the user for credentials by using a specified prompt window
/// caption, prompt message, user name and target name, credential
/// types allowed to be returned, and UI behavior options. In this
/// example this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <param name="caption">The caption for the message window.</param>
/// <param name="message">The text of the message.</param>
/// <param name="userName">The user name whose credential is to be
/// prompted for.</param>
/// <param name="targetName">The name of the target for which the
/// credential is collected.</param>
/// <param name="allowedCredentialTypes">A PSCredentialTypes constant
/// that identifies the type of credentials that can be returned.
</param>

```

```
/// <param name="options">A PSCredentialUIOptions constant that
/// identifies the UI behavior when it gathers the credentials.</param>
/// <returns>Throws a NotImplementedException exception.</returns>
public override PSCredential PromptForCredential(
    string caption,
    string message,
    string userName,
    string targetName,
    PSCredentialTypes
    allowedCredentialTypes,
    PSCredentialUIOptions options)
{
    throw new NotImplementedException(
        "The method or operation is not
implemented.");
}

/// <summary>
/// Reads characters that are entered by the user until a newline
/// (carriage return) is encountered.
/// </summary>
/// <returns>The characters that are entered by the user.</returns>
public override string ReadLine()
{
    return Console.ReadLine();
}

/// <summary>
/// Reads characters entered by the user until a newline (carriage
return)
/// is encountered and returns the characters as a secure string. In
this
/// example this functionality is not needed so the method throws a
/// NotImplementedException exception.
/// </summary>
/// <returns>Throws a NotImplementedException exception.</returns>
public override System.Security.SecureString ReadLineAsSecureString()
{
    throw new NotImplementedException("The method or operation is not
implemented.");
}

/// <summary>
/// Writes characters to the output display of the host.
/// </summary>
/// <param name="value">The characters to be written.</param>
public override void Write(string value)
{
    Console.Write(value);
}

/// <summary>
/// Writes characters to the output display of the host with possible
/// foreground and background colors.
/// </summary>
```

```
/// <param name="foregroundColor">The color of the characters.</param>
/// <param name="backgroundColor">The background color to use.</param>
/// <param name="value">The characters to be written.</param>
public override void Write(
    ConsoleColor foregroundColor,
    ConsoleColor backgroundColor,
    string value)
{
    ConsoleColor oldFg = Console.ForegroundColor;
    ConsoleColor oldBg = Console.BackgroundColor;
    Console.ForegroundColor = foregroundColor;
    Console.BackgroundColor = backgroundColor;
    Console.Write(value);
    Console.ForegroundColor = oldFg;
    Console.BackgroundColor = oldBg;
}

/// <summary>
/// Writes a line of characters to the output display of the host
/// with foreground and background colors and appends a newline
/// (carriage return).
/// </summary>
/// <param name="foregroundColor">The foreground color of the display.
</param>
/// <param name="backgroundColor">The background color of the display.
</param>
/// <param name="value">The line to be written.</param>
public override void WriteLine(
    ConsoleColor foregroundColor,
    ConsoleColor backgroundColor,
    string value)
{
    ConsoleColor oldFg = Console.ForegroundColor;
    ConsoleColor oldBg = Console.BackgroundColor;
    Console.ForegroundColor = foregroundColor;
    Console.BackgroundColor = backgroundColor;
    Console.WriteLine(value);
    Console.ForegroundColor = oldFg;
    Console.BackgroundColor = oldBg;
}

/// <summary>
/// Writes a debug message to the output display of the host.
/// </summary>
/// <param name="message">The debug message that is displayed.</param>
public override void WriteDebugLine(string message)
{
    this.WriteLine(
        ConsoleColor.DarkYellow,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "DEBUG: {0}",
message));
}

/// <summary>
```

```
/// Writes an error message to the output display of the host.
/// </summary>
/// <param name="value">The error message that is displayed.</param>
public override void WriteErrorLine(string value)
{
    this.WriteLine(
        ConsoleColor.Red,
        ConsoleColor.Black,
        value);
}

/// <summary>
/// Writes a newline character (carriage return)
/// to the output display of the host.
/// </summary>
public override void WriteLine()
{
    Console.WriteLine();
}

/// <summary>
/// Writes a line of characters to the output display of the host
/// and appends a newline character(carriage return).
/// </summary>
/// <param name="value">The line to be written.</param>
public override void WriteLine(string value)
{
    Console.WriteLine(value);
}

/// <summary>
/// Writes a progress report to the output display of the host.
/// </summary>
/// <param name="sourceId">Unique identifier of the source of the
record. </param>
/// <param name="record">A ProgressReport object.</param>
public override void WriteProgress(long sourceId, ProgressRecord record)
{

}

/// <summary>
/// Writes a verbose message to the output display of the host.
/// </summary>
/// <param name="message">The verbose message that is displayed.</param>
public override void WriteVerboseLine(string message)
{
    this.WriteLine(
        ConsoleColor.Green,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "VERBOSE:
{0}", message));
}

/// <summary>
```

```
/// Writes a warning message to the output display of the host.
/// </summary>
/// <param name="message">The warning message that is displayed.</param>
public override void WriteWarningLine(string message)
{
    this.WriteLine(
        ConsoleColor.Yellow,
        ConsoleColor.Black,
        String.Format(CultureInfo.CurrentCulture, "WARNING:
{0}", message));
}

/// <summary>
/// Parse a string containing a hotkey character.
/// Take a string of the form
///     Yes to &all
/// and returns a two-dimensional array split out as
///     "A", "Yes to all".
/// </summary>
/// <param name="input">The string to process</param>
/// <returns>
/// A two dimensional array containing the parsed components.
/// </returns>
private static string[] GetHotkeyAndLabel(string input)
{
    string[] result = new string[] { String.Empty, String.Empty };
    string[] fragments = input.Split('&');
    if (fragments.Length == 2)
    {
        if (fragments[1].Length > 0)
        {
            result[0] = fragments[1][0].ToString().
                ToUpper(CultureInfo.CurrentCulture);
        }

        result[1] = (fragments[0] + fragments[1]).Trim();
    }
    else
    {
        result[1] = input;
    }

    return result;
}

/// <summary>
/// This is a private worker function splits out the
/// accelerator keys from the menu and builds a two
/// dimensional array with the first access containing the
/// accelerator and the second containing the label string
/// with the & removed.
/// </summary>
/// <param name="choices">The choice collection to process</param>
/// <returns>
/// A two dimensional array containing the accelerator characters
```

```

/// and the cleaned-up labels
```

`private static string[,] BuildHotkeysAndPlainLabels(
 Collection<ChoiceDescription> choices)
{
 // Allocate the result array
 string[,] hotkeysAndPlainLabels = new string[2, choices.Count];

 for (int i = 0; i < choices.Count; ++i)
 {
 string[] hotkeyAndLabel = GetHotkeyAndLabel(choices[i].Label);
 hotkeysAndPlainLabels[0, i] = hotkeyAndLabel[0];
 hotkeysAndPlainLabels[1, i] = hotkeyAndLabel[1];
 }

 return hotkeysAndPlainLabels;
}
}
}
}`

Example 4

The following code is the implementation of the [System.Management.Automation.Host.PSHostRawUserInterface](#) class that is used by this host application. Those elements that are not implemented throw an exception or return nothing.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Management.Automation.Host;

    /// <summary>
    /// A sample implementation of the PSHostRawUserInterface for console
    /// applications. Members of this class that easily map to the .NET
    /// console class are implemented. More complex methods are not
    /// implemented and throw a NotImplementedException exception.
    /// </summary>
    internal class MyRawUserInterface : PSHostRawUserInterface
    {
        /// <summary>
        /// Gets or sets the background color of text to be written.
        /// This maps to the corresponding Console.BackgroundColor property.
        /// </summary>
        public override ConsoleColor BackgroundColor
        {
            get { return Console.BackgroundColor; }
            set { Console.BackgroundColor = value; }
        }
    }
}
```

```
/// <summary>
/// Gets or sets the host buffer size adapted from the Console buffer
/// size members.
/// </summary>
public override Size BufferSize
{
    get { return new Size(Console.BufferWidth, Console.BufferHeight); }
    set { Console.SetBufferSize(value.Width, value.Height); }
}

/// <summary>
/// Gets or sets the cursor position. In this example this
/// functionality is not needed so the property throws a
/// NotImplementedException exception.
/// </summary>
public override Coordinates CursorPosition
{
    get { throw new NotImplementedException(
        "The method or operation is not implemented."); }
    set { throw new NotImplementedException(
        "The method or operation is not implemented."); }
}

/// <summary>
/// Gets or sets the cursor size taken directly from the
/// Console.CursorSize property.
/// </summary>
public override int CursorSize
{
    get { return Console.CursorSize; }
    set { Console.CursorSize = value; }
}

/// <summary>
/// Gets or sets the foreground color of the text to be written.
/// This maps to the corresponding Console.ForegroundColor property.
/// </summary>
public override ConsoleColor ForegroundColor
{
    get { return Console.ForegroundColor; }
    set { Console.ForegroundColor = value; }
}

/// <summary>
/// Gets a value indicating whether a key is available. This maps to
/// the corresponding Console.KeyAvailable property.
/// </summary>
public override bool KeyAvailable
{
    get { return Console.KeyAvailable; }
}

/// <summary>
/// Gets the maximum physical size of the window adapted from the
/// Console.LargestWindowWidth and Console.LargestWindowHeight
```

```
/// properties.
/// </summary>
public override Size MaxPhysicalWindowSize
{
    get { return new Size(Console.LargestScreenWidth,
Console.LargestScreenHeight); }
}

/// <summary>
/// Gets the maximum window size adapted from the
/// Console.LargestScreenWidth and console.LargestScreenHeight
/// properties.
/// </summary>
public override Size MaxWindowSize
{
    get { return new Size(Console.LargestScreenWidth,
Console.LargestScreenHeight); }
}

/// <summary>
/// Gets or sets the window position adapted from the Console window
position
/// members.
/// </summary>
public override Coordinates WindowPosition
{
    get { return new Coordinates(Console.WindowLeft, Console.WindowTop); }
    set { Console.SetWindowPosition(value.X, value.Y); }
}

/// <summary>
/// Gets or sets the window size adapted from the corresponding Console
/// calls.
/// </summary>
public override Size WindowSize
{
    get { return new Size(Console.WindowWidth, Console.WindowHeight); }
    set { Console.SetWindowSize(value.Width, value.Height); }
}

/// <summary>
/// Gets or sets the title of the window mapped to the Console.Title
/// property.
/// </summary>
public override string WindowTitle
{
    get { return Console.Title; }
    set { Console.Title = value; }
}

/// <summary>
/// This API resets the input buffer. In this example this
/// functionality is not needed so the method returns nothing.
/// </summary>
public override void FlushInputBuffer()
```

```
{  
}  
  
/// <summary>  
/// This API returns a rectangular region of the screen buffer. In  
/// this example this functionality is not needed so the method throws  
/// a NotImplementedException exception.  
/// </summary>  
/// <param name="rectangle">Defines the size of the rectangle.</param>  
/// <returns>Throws a NotImplementedException exception.</returns>  
public override BufferCell[,] GetBufferContents(Rectangle rectangle)  
{  
    throw new NotImplementedException(  
        "The method or operation is not implemented.");  
}  
  
/// <summary>  
/// This API Reads a pressed, released, or pressed and released  
keystroke  
/// from the keyboard device, blocking processing until a keystroke is  
/// typed that matches the specified keystroke options. In this example  
/// this functionality is not needed so the method throws a  
/// NotImplementedException exception.  
/// </summary>  
/// <param name="options">Options, such as IncludeKeyDown, used when  
/// reading the keyboard.</param>  
/// <returns>Throws a NotImplementedException exception.</returns>  
public override KeyInfo ReadKey(ReadKeyOptions options)  
{  
    throw new NotImplementedException(  
        "The method or operation is not implemented.");  
}  
  
/// <summary>  
/// This API crops a region of the screen buffer. In this example  
/// this functionality is not needed so the method throws a  
/// NotImplementedException exception.  
/// </summary>  
/// <param name="source">The region of the screen to be scrolled.  
</param>  
/// <param name="destination">The region of the screen to receive the  
/// source region contents.</param>  
/// <param name="clip">The region of the screen to include in the  
operation.</param>  
/// <param name="fill">The character and attributes to be used to fill  
all cell.</param>  
public override void ScrollBufferContents(Rectangle source, Coordinates  
destination, Rectangle clip, BufferCell fill)  
{  
    throw new NotImplementedException(  
        "The method or operation is not implemented.");  
}  
  
/// <summary>  
/// This API copies an array of buffer cells into the screen buffer
```

```

    /// at a specified location. In this example this functionality is
    /// not needed so the method throws a NotImplementedException
exception.

    /// </summary>
    /// <param name="origin">The parameter is not used.</param>
    /// <param name="contents">The parameter is not used.</param>
    public override void SetBufferContents(Coordinates origin, BufferCell[,] contents)
    {
        throw new NotImplementedException(
            "The method or operation is not implemented.");
    }

    /// <summary>
    /// This API Copies a given character, foreground color, and background
    /// color to a region of the screen buffer. In this example this
    /// functionality is not needed so the method throws a
    /// NotImplementedException exception./// </summary>
    /// <param name="rectangle">Defines the area to be filled. </param>
    /// <param name="fill">Defines the fill character.</param>
    public override void SetBufferContents(Rectangle rectangle, BufferCell fill)
    {
        throw new NotImplementedException(
            "The method or operation is not implemented.");
    }
}

```

Example 5

The following code reads the command line and colors the text as it is entered. Tokens are determined by using the [System.Management.Automation.PSParser.Tokenize*](#) method.

C#

```

namespace Microsoft.Samples.PowerShell.Host
{
    using System;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Text;

    /// <summary>
    /// This class is used to read the command line and color the text as
    /// it is entered. Tokens are determined using the PSParser.Tokenize
    /// method.
    /// </summary>
    internal class ConsoleReadLine
    {

```

```
/// <summary>
/// The buffer used to edit.
/// </summary>
private StringBuilder buffer = new StringBuilder();

/// <summary>
/// The position of the cursor within the buffer.
/// </summary>
private int current;

/// <summary>
/// The count of characters in buffer rendered.
/// </summary>
private int rendered;

/// <summary>
/// Store the anchor and handle cursor movement
/// </summary>
private Cursor cursor;

/// <summary>
/// The array of colors for tokens, indexed by PSTokenType
/// </summary>
private ConsoleColor[] tokenColors;

/// <summary>
/// We do not pick different colors for every token, those tokens
/// use this default.
/// </summary>
private ConsoleColor defaultColor = Console.ForegroundColor;

/// <summary>
/// Initializes a new instance of the Console.ReadLine class.
/// </summary>
public Console.ReadLine()
{
    this.tokenColors = new ConsoleColor[]
    {
        this.defaultColor,           // Unknown
        ConsoleColor.Yellow,        // Command
        ConsoleColor.Green,         // CommandParameter
        ConsoleColor.Cyan,          // CommandArgument
        ConsoleColor.Cyan,          // Number
        ConsoleColor.Cyan,          // String
        ConsoleColor.Green,          // Variable
        this.defaultColor,          // Member
        this.defaultColor,          // LoopLabel
        ConsoleColor.DarkYellow,     // Attribute
        ConsoleColor.DarkYellow,     // Type
        ConsoleColor.DarkCyan,       // Operator
        this.defaultColor,          // GroupStart
        this.defaultColor,          // GroupEnd
        ConsoleColor.Magenta,        // Keyword
        ConsoleColor.Red,           // Comment
        ConsoleColor.DarkCyan,       // StatementSeparator
    };
}
```

```
        this.defaultColor,           // NewLine
        this.defaultColor,           // LineContinuation
        this.defaultColor,           // Position
    };
}

/// <summary>
/// Read a line of text, colorizing while typing.
/// </summary>
/// <returns>The command line read</returns>
public string Read()
{
    this.Initialize();

    while (true)
    {
        ConsoleKeyInfo key = Console.ReadKey(true);

        switch (key.Key)
        {
            case ConsoleKey.Backspace:
                this.OnBackspace();
                break;
            case ConsoleKey.Delete:
                this.OnDelete();
                break;
            case ConsoleKey.Enter:
                return this.OnEnter();
            case ConsoleKey.RightArrow:
                this.OnRight(key.Modifiers);
                break;
            case ConsoleKey.LeftArrow:
                this.OnLeft(key.Modifiers);
                break;
            case ConsoleKey.Escape:
                this.OnEscape();
                break;
            case ConsoleKey.Home:
                this.OnHome();
                break;
            case ConsoleKey.End:
                this.OnEnd();
                break;
            case ConsoleKey.UpArrow:
            case ConsoleKey.DownArrow:
            case ConsoleKey.LeftWindows:
            case ConsoleKey.RightWindows:
                // ignore these
                continue;

            default:
                if (key.KeyChar == '\x0D')
                {
                    goto case ConsoleKey.Enter;      // Ctrl-M
                }
        }
    }
}
```

```
        if (key.KeyChar == '\x08')
        {
            goto case ConsoleKey.Backspace; // Ctrl-H
        }

        this.Insert(key);
        break;
    }
}

/// <summary>
/// Initializes the buffer.
/// </summary>
private void Initialize()
{
    this.buffer.Length = 0;
    this.current = 0;
    this.rendered = 0;
    this.cursor = new Cursor();
}

/// <summary>
/// Inserts a key.
/// </summary>
/// <param name="key">The key to insert.</param>
private void Insert(ConsoleKeyInfo key)
{
    this.buffer.Insert(this.current, key.KeyChar);
    this.current++;
    this.Render();
}

/// <summary>
/// The End key was entered..
/// </summary>
private void OnEnd()
{
    this.current = this.buffer.Length;
    this.cursor.Place(this.rendered);
}

/// <summary>
/// The Home key was entered.
/// </summary>
private void OnHome()
{
    this.current = 0;
    this.cursor.Reset();
}

/// <summary>
/// The Escape key was entered.
/// </summary>
```

```

private void OnEscape()
{
    this.buffer.Length = 0;
    this.current = 0;
    this.Render();
}

/// <summary>
/// Moves to the left of the cursor position.
/// </summary>
/// <param name="consoleModifiers">Enumeration for Alt, Control,
/// and Shift keys.</param>
private void OnLeft(ConsoleModifiers consoleModifiers)
{
    if ((consoleModifiers & ConsoleModifiers.Control) != 0)
    {
        // Move back to the start of the previous word.
        if (this.buffer.Length > 0 && this.current != 0)
        {
            bool nonLetter = IsSeparator(this.buffer[this.current - 1]);
            while (this.current > 0 && (this.current - 1 <
this.buffer.Length))
            {
                this.MoveLeft();

                if (IsSeparator(this.buffer[this.current])) != nonLetter)
                {
                    if (!nonLetter)
                    {
                        this.MoveRight();
                        break;
                    }

                    nonLetter = false;
                }
            }
        }
    }
    else
    {
        this.MoveLeft();
    }
}

/// <summary>
/// Determines if a character is a separator.
/// </summary>
/// <param name="ch">Character to investigate.</param>
/// <returns>A value that indicates whether the character
/// is a separator.</returns>
private static bool IsSeparator(char ch)
{
    return !Char.IsLetter(ch);
}

```

```

/// <summary>
/// Moves to what is to the right of the cursor position.
/// </summary>
/// <param name="consoleModifiers">Enumeration for Alt, Control,
/// and Shift keys.</param>
private void OnRight(ConsoleModifiers consoleModifiers)
{
    if ((consoleModifiers & ConsoleModifiers.Control) != 0)
    {
        // Move to the next word.
        if (this.buffer.Length != 0 && this.current < this.buffer.Length)
        {
            bool nonLetter = IsSeparator(this.buffer[this.current]);
            while (this.current < this.buffer.Length)
            {
                this.MoveRight();

                if (this.current == this.buffer.Length)
                {
                    break;
                }

                if (IsSeparator(this.buffer[this.current]) != nonLetter)
                {
                    if (nonLetter)
                    {
                        break;
                    }

                    nonLetter = true;
                }
            }
        }
        else
        {
            this.MoveRight();
        }
    }
}

/// <summary>
/// Moves the cursor one character to the right.
/// </summary>
private void MoveRight()
{
    if (this.current < this.buffer.Length)
    {
        char c = this.buffer[this.current];
        this.current++;
        Cursor.Move(1);
    }
}

/// <summary>
/// Moves the cursor one character to the left.

```

```
/// </summary>
private void MoveLeft()
{
    if (this.current > 0 && (this.current - 1 < this.buffer.Length))
    {
        this.current--;
        char c = this.buffer[this.current];
        Cursor.Move(-1);
    }
}

/// <summary>
/// The Enter key was entered.
/// </summary>
/// <returns>A newline character.</returns>
private string OnEnter()
{
    Console.Out.WriteLine("\n");
    return this.buffer.ToString();
}

/// <summary>
/// The delete key was entered.
/// </summary>
private void OnDelete()
{
    if (this.buffer.Length > 0 && this.current < this.buffer.Length)
    {
        this.buffer.Remove(this.current, 1);
        this.Render();
    }
}

/// <summary>
/// The Backspace key was entered.
/// </summary>
private void OnBackspace()
{
    if (this.buffer.Length > 0 && this.current > 0)
    {
        this.buffer.Remove(this.current - 1, 1);
        this.current--;
        this.Render();
    }
}

/// <summary>
/// Displays the line.
/// </summary>
private void Render()
{
    string text = this.buffer.ToString();

    // The PowerShell tokenizer is used to decide how to colorize
    // the input. Any errors in the input are returned in 'errors',
}
```

```

// but we won't be looking at those here.
Collection<PSParseError> errors = null;
Collection<PSToken> tokens = PSParser.Tokenize(text, out errors);

if (tokens.Count > 0)
{
    // We can skip rendering tokens that end before the cursor.
    int i;
    for (i = 0; i < tokens.Count; ++i)
    {
        if (this.current >= tokens[i].Start)
        {
            break;
        }
    }

    // Place the cursor at the start of the first token to render. The
    // last edit may require changes to the colorization of characters
    // preceding the cursor.
    this.cursor.Place(tokens[i].Start);

    for (; i < tokens.Count; ++i)
    {
        // Write out the token. We don't use tokens[i].Content, instead
        we
        // use the actual text from our input because the content
        sometimes
        // excludes part of the token, e.g. the quote characters of a
        string.
        Console.ForegroundColor = this.tokenColors[(int)tokens[i].Type];
        Console.Out.Write(text.Substring(tokens[i].Start,
tokens[i].Length));

        // Whitespace doesn't show up in the array of tokens. Write it
        out here.
        if (i != (tokens.Count - 1))
        {
            Console.ForegroundColor = this.defaultColor;
            for (int j = (tokens[i].Start + tokens[i].Length); j < tokens[i
+ 1].Start; ++j)
            {
                Console.Out.Write(text[j]);
            }
        }
    }

    // It's possible there is text left over to output. This happens
    when there is
    // some error during tokenization, e.g. a string literal is missing
    a closing quote.
    Console.ForegroundColor = this.defaultColor;
    for (int j = tokens[i - 1].Start + tokens[i - 1].Length; j <
text.Length; ++j)
    {
        Console.Out.Write(text[j]);
    }
}

```

```
        }
    }
    else
    {
        // If tokenization completely failed, just redraw the whole line.
This
        // happens most frequently when the first token is incomplete, like
a string
        // literal missing a closing quote.
        this.cursor.Reset();
        Console.Out.Write(text);
    }

    // If characters were deleted, we must write over previously written
characters
    if (text.Length < this.rendered)
    {
        Console.Out.Write(new string(' ', this.rendered - text.Length));
    }

    this.rendered = text.Length;
    this.cursor.Place(this.current);
}

/// <summary>
/// A helper class for maintaining the cursor while editing the command
line.
/// </summary>
internal class Cursor
{
    /// <summary>
    /// The top anchor for repositioning the cursor.
    /// </summary>
    private int anchorTop;

    /// <summary>
    /// The left anchor for repositioning the cursor.
    /// </summary>
    private int anchorLeft;

    /// <summary>
    /// Initializes a new instance of the Cursor class.
    /// </summary>
    public Cursor()
    {
        this.anchorTop = Console.CursorTop;
        this.anchorLeft = Console.CursorLeft;
    }

    /// <summary>
    /// Moves the cursor.
    /// </summary>
    /// <param name="delta">The number of characters to move.</param>
    internal static void Move(int delta)
{
```

```

        int position = Console.CursorTop * Console.BufferWidth +
Console.CursorLeft + delta;

        Console.CursorLeft = position % Console.BufferWidth;
        Console.CursorTop = position / Console.BufferWidth;
    }

    /// <summary>
    /// Resets the cursor position.
    /// </summary>
    internal void Reset()
    {
        Console.CursorTop = this.anchorTop;
        Console.CursorLeft = this.anchorLeft;
    }

    /// <summary>
    /// Moves the cursor to a specific position.
    /// </summary>
    /// <param name="position">The new position.</param>
    internal void Place(int position)
    {
        Console.CursorLeft = (this.anchorLeft + position) %
Console.BufferWidth;
        int cursorTop = this.anchorTop + (this.anchorLeft + position) /
Console.BufferWidth;
        if (cursorTop >= Console.BufferHeight)
        {
            this.anchorTop -= cursorTop - Console.BufferHeight + 1;
            cursorTop = Console.BufferHeight - 1;
        }

        Console.CursorTop = cursorTop;
    }
} // End Cursor
}
}

```

See Also

[System.Management.Automation.Host.PSHost](#)

[System.Management.Automation.Host.PSHostUserInterface](#)

[System.Management.Automation.Host.PSHostRawUserInterface](#)

Runspace Samples

Article • 03/24/2025

This section includes sample code that shows how to use different types of runspaces to run commands synchronously and asynchronously. You can use Microsoft Visual Studio to create a console application and then copy the code from the topics in this section into your host application.

In This Section

Note

For samples of host applications that create custom host interfaces, see [Custom Host Samples](#).

[Runspace01 Sample](#) This sample shows how to use the `System.Management.Automation.PowerShell` class to run the `Get-Process` cmdlet synchronously and display its output in a console window.

[Runspace02 Sample](#) This sample shows how to use the `System.Management.Automation.PowerShell` class to run the `Get-Process` and `Sort-Object` cmdlets synchronously. The results of these commands is displayed by using a `System.Windows.Forms.DataGridView` control.

[Runspace03 Sample](#) This sample shows how to use the `System.Management.Automation.PowerShell` class to run a script synchronously, and how to handle non-terminating errors. The script receives a list of process names and then retrieves those processes. The results of the script, including any non-terminating errors that were generated when running the script, are displayed in a console window.

[Runspace04 Sample](#) This sample shows how to use the `System.Management.Automation.PowerShell` class to run commands, and how to catch terminating errors that are thrown when running the commands. Two commands are run, and the last command is passed a parameter argument that is not valid. As a result no objects are returned and a terminating error is thrown.

[Runspace05 Sample](#) This sample shows how to add a snap-in to a `System.Management.Automation.Runspaces.InitialSessionState` object so that the cmdlet of the snap-in is available when the runspace is opened. The snap-in provides a

Get-Proc cmdlet (defined by the [GetProcessSample01 Sample](#)) that is run synchronously using a [System.Management.Automation.PowerShell](#) object.

Runspace06 Sample This sample shows how to add a module to a [System.Management.Automation.Runspaces.InitialSessionState](#) object so that the module is loaded when the runspace is opened. The module provides a Get-Proc cmdlet (defined by the [GetProcessSample02 Sample](#)) that is run synchronously using a [System.Management.Automation.PowerShell](#) object.

Runspace07 Sample This sample shows how to create a runspace, and then use that runspace to run two cmdlets synchronously by using a [System.Management.Automation.PowerShell](#) object.

Runspace08 Sample This sample shows how to add commands and arguments to the pipeline of a [System.Management.Automation.PowerShell](#) object and how to run the commands synchronously.

Runspace09 Sample This sample shows how to add a script to the pipeline of a [System.Management.Automation.PowerShell](#) object and how to run the script asynchronously. Events are used to handle the output of the script.

Runspace10 Sample This sample shows how to create a default initial session state, how to add a cmdlet to the [System.Management.Automation.Runspaces.InitialSessionState](#), how to create a runspace that uses the initial session state, and how to run the command by using a [System.Management.Automation.PowerShell](#) object.

Runspace11 Sample This shows how to use the [System.Management.Automation.ProxyCommand](#) class to create a proxy command that calls an existing cmdlet, but restricts the set of available parameters. The proxy command is then added to an initial session state that is used to create a constrained runspace. This means that the user can access the functionality of the cmdlet only through the proxy command.

See Also

Runspace01 Sample

Article • 09/17/2021

This sample shows how to use the [System.Management.Automation.PowerShell](#) class to run the [Get-Process](#) cmdlet synchronously. The [Get-Process](#) cmdlet returns [System.Diagnostics.Process](#) objects for each process running on the local computer. The values of the [System.Diagnostics.Process.ProcessName*](#) and [System.Diagnostics.Process.HandleCount*](#) properties are then extracted from the returned objects and displayed in a console window.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

- Creating a [System.Management.Automation.PowerShell](#) object to run a command.
- Adding a command to the pipeline of the [System.Management.Automation.PowerShell](#) object.
- Running the command synchronously.
- Using [System.Management.Automation.PSObject](#) objects to extract properties from the objects returned by the command.

Example

This sample runs the [Get-Process](#) cmdlet synchronously in the default runspace provided by Windows PowerShell.

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Management.Automation;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace01
```

```

{
    /// <summary>
    /// This sample uses the PowerShell class to execute
    /// the Get-Process cmdlet synchronously. The name and
    /// handlecount are then extracted from the PSObjects
    /// returned and displayed.
    /// </summary>
    /// <param name="args">Parameter not used.</param>
    /// <remarks>
    /// This sample demonstrates the following:
    /// 1. Creating a PowerShell object to run a command.
    /// 2. Adding a command to the pipeline of the PowerShell object.
    /// 3. Running the command synchronously.
    /// 4. Using PSObject objects to extract properties from the objects
    ///     returned by the command.
    /// </remarks>
    private static void Main(string[] args)
    {
        // Create a PowerShell object. Creating this object takes care of
        // building all of the other data structures needed to run the
        command.

        using (PowerShell powershell = PowerShell.Create().AddCommand("Get-
Process"))
        {
            Console.WriteLine("Process                  HandleCount");
            Console.WriteLine("-----");

            // Invoke the command synchronously and display the
            // ProcessName and HandleCount properties of the
            // objects that are returned.
            foreach (PSObject result in powershell.Invoke())
            {
                Console.WriteLine(
                    "{0,-20} {1}",
                    result.Members["ProcessName"].Value,
                    result.Members["HandleCount"].Value);
            }
        }

        System.Console.WriteLine("Hit any key to exit...");
        System.Console.ReadKey();
    }
}

```

See Also

Runspace02 Sample

Article • 03/24/2025

This sample shows how to use the [System.Management.Automation.PowerShell](#) class to run the [Get-Process](#) and [Sort-Object](#) cmdlets synchronously. The [Get-Process](#) cmdlet returns [System.Diagnostics.Process](#) objects for each process running on the local computer, and the [Sort-Object](#) sorts the objects based on their [System.Diagnostics.Process.Id*](#) property. The results of these commands is displayed by using a [System.Windows.Forms.DataGridView](#) control.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Creating a [System.Management.Automation.PowerShell](#) object to run commands.
- Adding commands to the pipeline of [System.Management.Automation.PowerShell](#) object.
- Running the commands synchronously.
- Using a [System.Windows.Forms.DataGridView](#) control to display the output of the commands in a Windows Forms application.

Example

This sample runs the [Get-Process](#) and [Sort-Object](#) cmdlets synchronously in the default runspace provided by Windows PowerShell. The output is displayed in a form using a [System.Windows.Forms.DataGridView](#) control.

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
```

```
using System.Management.Automation.Runspaces;
using System.Windows.Forms;
using PowerShell = System.Management.Automation.PowerShell;

/// <summary>
/// This class contains the Main entry point for this host application.
/// </summary>
internal class Runspace02
{
    /// <summary>
    /// This method creates the form where the output is displayed.
    /// </summary>
    private static void CreateForm()
    {
        Form form = new Form();
        DataGridView grid = new DataGridView();
        form.Controls.Add(grid);
        grid.Dock = DockStyle.Fill;

        // Create a PowerShell object. Creating this object takes care of
        // building all of the other data structures needed to run the
        command.

        using (PowerShell powershell = PowerShell.Create())
        {
            powershell.AddCommand("Get-Process").AddCommand("Sort-
Object").AddArgument("ID");
            if (Runspace.DefaultRunspace == null)
            {
                Runspace.DefaultRunspace = powershell.Runspace;
            }

            Collection<PSObject> results = powershell.Invoke();

            // The generic collection needs to be re-wrapped in an ArrayList
            // for data-binding to work.
            ArrayList objects = new ArrayList();
            objects.AddRange(results);

            // The DataGridView will use the PSObjectTypeDescriptor type
            // to retrieve the properties.
            grid.DataSource = objects;
        }

        form.ShowDialog();
    }

    /// <summary>
    /// This sample uses a PowerShell object to run the Get-Process
    /// and Sort-Object cmdlets synchronously. Windows Forms and
    /// data binding are then used to display the results in a
    /// DataGridView control.
    /// </summary>
    /// <param name="args">The parameter is not used.</param>
    /// <remarks>
    /// This sample demonstrates the following:
    /// </remarks>
}
```

```
/// 1. Creating a PowerShell object.  
/// 2. Adding commands and arguments to the pipeline of  
///     the PowerShell object.  
/// 3. Running the commands synchronously.  
/// 4. Using a DataGridView control to display the output  
///     of the commands in a Windows Forms application.  
/// </remarks>  
private static void Main(string[] args)  
{  
    Runspace02.CreateForm();  
}  
}  
}
```

See Also

[Writing a Windows PowerShell Host Application](#)

Runspace03 Sample

Article • 09/17/2021

This sample shows how to use the [System.Management.Automation.PowerShell](#) class to run a script synchronously, and how to handle non-terminating errors. The script receives a list of process names and then retrieves those processes. The results of the script, including any non-terminating errors that were generated when running the script, are displayed in a console window.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Creating a [System.Management.Automation.PowerShell](#) object to run a script.
- Adding a script to the pipeline of the [System.Management.Automation.PowerShell](#) object.
- Passing input objects to the script from the calling program.
- Running the script synchronously.
- Using [System.Management.Automation.PSObject](#) objects to extract and display properties from the objects returned by the script.
- Retrieving and displaying error records that were generated when the script was run.

Example

This sample runs a script synchronously in the default runspace provided by Windows PowerShell. The output of the script and any non-terminating errors that were generated are displayed in a console window.

```
C#
```

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
```

```
using System;
using System.Collections;
using System.Management.Automation;
using System.Management.Automation.Runspaces;
using PowerShell = System.Management.Automation.PowerShell;

/// <summary>
/// This class contains the Main entry point for this host application.
/// </summary>
internal class Runspace03
{
    /// <summary>
    /// This sample shows how to use the PowerShell class to run a
    /// script that retrieves process information for the list of
    /// process names passed to the script. It shows how to pass input
    /// objects to a script and how to retrieve error objects as well
    /// as the output objects.
    /// </summary>
    /// <param name="args">Parameter not used.</param>
    /// <remarks>
    /// This sample demonstrates the following:
    /// 1. Creating a PowerShell object to run a script.
    /// 2. Adding a script to the pipeline of the PowerShell object.
    /// 3. Passing input objects to the script from the calling program.
    /// 4. Running the script synchronously.
    /// 5. Using PSObject objects to extract and display properties from
    ///     the objects returned by the script.
    /// 6. Retrieving and displaying error records that were generated
    ///     when the script was run.
    /// </remarks>
    private static void Main(string[] args)
    {
        // Define a list of processes to look for.
        string[] processNames = new string[]
        {
            "lsass", "nosuchprocess", "services", "nosuchprocess2"
        };

        // The script to run to get these processes. Input passed
        // to the script will be available in the $input variable.
        string script = "$input | Get-Process -Name ${_}";

        // Create a PowerShell object. Creating this object takes care of
        // building all of the other data structures needed to run the script.
        using (PowerShell powershell = PowerShell.Create())
        {
            powershell.AddScript(script);

            Console.WriteLine("Process                  HandleCount");
            Console.WriteLine("-----");

            // Invoke the script synchronously and display the
            // ProcessName and HandleCount properties of the
            // objects that are returned.
            foreach (PSObject result in powershell.Invoke(processNames))

```

```
        {
            Console.WriteLine(
                "{0,-20} {1}",
                result.Members["ProcessName"].Value,
                result.Members["HandleCount"].Value);
        }

        // Process any error records that were generated while running
        // the script.
        Console.WriteLine("\nThe following non-terminating errors
occurred:\n");
        PSDataCollection<ErrorRecord> errors = powershell.Streams.Error;
        if (errors != null && errors.Count > 0)
        {
            foreach (ErrorRecord err in errors)
            {
                System.Console.WriteLine("    error: {0}", err.ToString());
            }
        }
    }

    System.Console.WriteLine("\nHit any key to exit...");
    System.Console.ReadKey();
}
}
```

See Also

[Writing a Windows PowerShell Host Application](#)

Runspace04 Sample

Article • 09/17/2021

This sample shows how to use the [System.Management.Automation.PowerShell](#) class to run commands, and how to catch terminating errors that are thrown when running the commands. Two commands are run, and the last command is passed a parameter argument that is not valid. As a result, no objects are returned and a terminating error is thrown.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Creating a [System.Management.Automation.PowerShell](#) object.
- Adding commands to the pipeline of the [System.Management.Automation.PowerShell](#) object.
- Adding parameter arguments to the pipeline.
- Invoking the commands synchronously.
- Using [System.Management.Automation.PSObject](#) objects to extract and display properties from the objects returned by the commands.
- Retrieving and displaying error records that were generated during the running of the commands.
- Catching and displaying terminating exceptions thrown by the commands.

Example

This sample runs commands synchronously in the default runspace provided by Windows PowerShell. The last command throws a terminating error because a parameter argument that is not valid is passed to the command. The terminating error is trapped and displayed.

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace04
    {
        /// <summary>
        /// This sample shows how to use a PowerShell object to run commands.
        /// The commands generate a terminating exception that the caller
        /// should catch and process.
        /// </summary>
        /// <param name="args">The parameter is not used.</param>
        /// <remarks>
        /// This sample demonstrates the following:
        /// 1. Creating a PowerShell object to run commands.
        /// 2. Adding commands to the pipeline of the PowerShell object.
        /// 3. Passing input objects to the commands from the calling program.
        /// 4. Using PSObject objects to extract and display properties from the
        ///     objects returned by the commands.
        /// 5. Retrieving and displaying error records that were generated
        ///     while running the commands.
        /// 6. Catching and displaying terminating exceptions generated
        ///     while running the commands.
        /// </remarks>
        private static void Main(string[] args)
        {
            // Create a PowerShell object.
            using (PowerShell powershell = PowerShell.Create())
            {
                // Add the commands to the PowerShell object.
                powershell.AddCommand("Get-ChildItem").AddCommand("Select-
String").AddArgument("*");

                // Run the commands synchronously. Because of the bad regular
                expression,
                // no objects will be returned. Instead, an exception will be
                thrown.
                try
                {
                    foreach (PSObject result in powershell.Invoke())
                    {
                        Console.WriteLine('{0}', result.ToString());
                    }
                }

                // Process any error records that were generated while running the
                commands.
            }
        }
    }
}
```

```
Console.WriteLine("\nThe following non-terminating errors
occurred:\n");
PSDataCollection<ErrorRecord> errors = powershell.Streams.Error;
if (errors != null && errors.Count > 0)
{
    foreach (ErrorRecord err in errors)
    {
        System.Console.WriteLine("    error: {0}", err.ToString());
    }
}
catch (RuntimeException runtimeException)
{
    // Trap any exception generated by the commands. These exceptions
    // will all be derived from the RuntimeException exception.
    System.Console.WriteLine(
        "Runtime exception: {0}: {1}\n{2}",
        runtimeException.ErrorRecord.InvocationInfo.InvocationName,
        runtimeException.Message,
        runtimeException.ErrorRecord.InvocationInfo.PositionMessage);
}
}

System.Console.WriteLine("\nHit any key to exit...");
System.Console.ReadKey();
}
}
}
```

See Also

[Writing a Windows PowerShell Host Application](#)

Runspace05 Sample

Article • 03/24/2025

This sample shows how to add a snap-in to a [System.Management.Automation.Runspaces.InitialSessionState](#) object so that the cmdlet of the snap-in is available when the runspace is opened. The snap-in provides a Get-Proc cmdlet (defined by the [GetProcessSample01 Sample](#)) that is run synchronously by using a [System.Management.Automation.PowerShell](#) object.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Creating a [System.Management.Automation.Runspaces.InitialSessionState](#) object.
- Adding the snap-in to the [System.Management.Automation.Runspaces.InitialSessionState](#) object.
- Creating a [System.Management.Automation.Runspaces.Runspace](#) object that uses the [System.Management.Automation.Runspaces.InitialSessionState](#) object.
- Creating a [System.Management.Automation.PowerShell](#) object that uses the runspace.
- Adding the snap-in's Get-Proc cmdlet to the pipeline of the [System.Management.Automation.PowerShell](#) object.
- Running the command synchronously.
- Extracting properties from the [System.Management.Automation.PSObject](#) objects returned by the command.

Example

This sample creates a runspace that uses a [System.Management.Automation.Runspaces.InitialSessionState](#) object to define the

elements that are available when the runspace is opened. In this sample, a snap-in that defines a Get-Proc cmdlet is added to the initial session state.

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace05
    {
        /// <summary>
        /// This sample shows how to define an initial session state that is
        /// used when creating a runspace. The sample invokes a command from
        /// a Windows PowerShell snap-in that is present in the console file.
        /// </summary>
        /// <param name="args">The parameter is not used.</param>
        /// <remarks>
        /// This sample assumes that user has copied the GetProcessSample01.dll
        /// that is produced by the GetProcessSample01 sample to the current
        /// directory.
        /// This sample demonstrates the following:
        /// 1. Creating a default initial session state.
        /// 2. Adding a snap-in to the initial session state.
        /// 3. Creating a runspace that uses the initial session state.
        /// 4. Creating a PowerShell object that uses the runspace.
        /// 5. Adding the snap-in's Get-Proc cmdlet to the PowerShell object.
        /// 6. Using PSObject objects to extract and display properties from
        ///     the objects returned by the cmdlet.
        /// </remarks>
        private static void Main(string[] args)
        {
            // Create the default initial session state. The default initial
            // session state contains all the elements provided by Windows
            // PowerShell.
            InitialSessionState iss = InitialSessionState.CreateDefault();
            PSSnapInException warning;
            iss.ImportPSSnapIn("GetProcPSSnapIn01", out warning);

            // Create a runspace. Notice that no PSHost object is supplied to the
            // CreateRunspace method so the default host is used. See the Host
            // samples for more information on creating your own custom host.
            using (Runspace myRunSpace = RunspaceFactory.CreateRunspace(iss))
            {
                myRunSpace.Open();

                // Create a PowerShell object.
```

```
using (PowerShell powershell = PowerShell.Create())
{
    // Add the snap-in cmdlet and specify the runspace.
    powershell.AddCommand("GetProcPSSnapIn01\Get-Proc");
    powershell.Runspace = myRunSpace;

    // Run the cmdlet synchronously.
    Collection<PSObject> results = powershell.Invoke();

    Console.WriteLine("Process           HandleCount");
    Console.WriteLine("-----");

    // Display the results.
    foreach (PSObject result in results)
    {
        Console.WriteLine(
            "{0,-20} {1}",
            result.Members["ProcessName"].Value,
            result.Members["HandleCount"].Value);
    }
}

// Close the runspace to release any resources.
myRunSpace.Close();
}
System.Console.WriteLine("Hit any key to exit...");
System.Console.ReadKey();
}
}
```

See Also

[Writing a Windows PowerShell Host Application](#)

Runspace06 Sample

Article • 03/24/2025

This sample shows how to add a module to a [System.Management.Automation.Runspaces.InitialSessionState](#) object so that the module is loaded when the runspace is opened. The module provides a Get-Proc cmdlet (defined by the [GetProcessSample02 Sample](#)) that is run synchronously by using a [System.Management.Automation.PowerShell](#) object.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Creating a [System.Management.Automation.Runspaces.InitialSessionState](#) object.
- Adding the module to the [System.Management.Automation.Runspaces.InitialSessionState](#) object.
- Creating a [System.Management.Automation.Runspaces.Runspace](#) object that uses the [System.Management.Automation.Runspaces.InitialSessionState](#) object.
- Creating a [System.Management.Automation.PowerShell](#) object that uses the runspace.
- Adding the module's Get-Proc cmdlet to the pipeline of the [System.Management.Automation.PowerShell](#) object.
- Running the command synchronously.
- Extracting properties from the [System.Management.Automation.PSObject](#) objects returned by the command.

Example

This sample creates a runspace that uses a [System.Management.Automation.Runspaces.InitialSessionState](#) object to define the

elements that are available when the runspace is opened. In this sample, a module that defines a Get-Proc cmdlet is added to the initial session state.

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace06
    {
        /// <summary>
        /// This sample shows how to define an initial session state that is
        /// used when creating a runspace. The sample invokes a command from
        /// a binary module that is loaded by the initial session state.
        /// </summary>
        /// <param name="args">Parameter not used.</param>
        /// <remarks>
        /// This sample assumes that user has copied the GetProcessSample02.dll
        /// that is produced by the GetProcessSample02 sample to the current
        /// directory.
        /// This sample demonstrates the following:
        /// 1. Creating a default initial session state.
        /// 2. Adding a module to the initial session state.
        /// 3. Creating a runspace that uses the initial session state.
        /// 4. Creating a PowerShell object that uses the runspace.
        /// 5. Adding the module's Get-Proc cmdlet to the PowerShell object.
        /// 6. Running the command synchronously.
        /// 7. Using PSObject objects to extract and display properties from
        ///     the objects returned by the cmdlet.
        /// </remarks>
        private static void Main(string[] args)
        {
            // Create the default initial session state and add the module.
            InitialSessionState iss = InitialSessionState.CreateDefault();
            iss.ImportPSModule(new string[] { @".\GetProcessSample02.dll" });

            // Create a runspace. Notice that no PSHost object is supplied to the
            // CreateRunspace method so the default host is used. See the Host
            // samples for more information on creating your own custom host.
            using (Runspace myRunSpace = RunspaceFactory.CreateRunspace(iss))
            {
                myRunSpace.Open();

                // Create a PowerShell object.
                using (PowerShell powershell = PowerShell.Create())
                {
```

```
// Add the cmdlet and specify the runspace.  
powershell.AddCommand(@"GetProcessSample02\Get-Proc");  
powershell.Runspace = myRunSpace;  
  
Collection<PSObject> results = powershell.Invoke();  
  
Console.WriteLine("Process           HandleCount");  
Console.WriteLine("-----");  
  
// Display the results.  
foreach (PSObject result in results)  
{  
    Console.WriteLine(  
        "{0,-20} {1}",  
        result.Members["ProcessName"].Value,  
        result.Members["HandleCount"].Value);  
}  
}  
  
// Close the runspace to release any resources.  
myRunSpace.Close();  
}  
  
System.Console.WriteLine("Hit any key to exit...");  
System.Console.ReadKey();  
}  
}  
}
```

See Also

[Writing a Windows PowerShell Host Application](#)

Runspace07 Sample

Article • 03/24/2025

This sample shows how to create a runspace, and then use that runspace to run two cmdlets synchronously by using a [System.Management.Automation.PowerShell](#) object.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Creating a [System.Management.Automation.Runspaces.Runspace](#) object by using the [System.Management.Automation.Runspaces.RunspaceFactory](#) class.
- Creating a [System.Management.Automation.PowerShell](#) object that uses the runspace.
- Adding cmdlets to the pipeline of the [System.Management.Automation.PowerShell](#) object.
- Running the cmdlets synchronously.
- Extracting properties from the [System.Management.Automation.PSObject](#) objects returned by the command.

Example

This sample creates a runspace that used by a [System.Management.Automation.PSObject](#) object to run the [Get-Process](#) and [Measure-Object](#) cmdlets.

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;
```

```
/// <summary>
/// This class contains the Main entry point for this host application.
/// </summary>
internal class Runspace07
{
    /// <summary>
    /// This sample shows how to create a runspace and how to run commands
    /// using a PowerShell object. It builds a pipeline that runs the
    /// Get-Process cmdlet, which is piped to the Measure-Object
    /// cmdlet to count the number of processes running on the system.
    /// </summary>
    /// <param name="args">The parameter is not used.</param>
    /// <remarks>
    /// This sample demonstrates the following:
    /// 1. Creating a runspace using the RunspaceFactory class.
    /// 2. Creating a PowerShell object that uses the runspace.
    /// 3. Adding cmdlets to the pipeline of the PowerShell object.
    /// 4. Running the cmdlets synchronously.
    /// 5. Working with PSObject objects to extract properties
    ///     from the objects returned by the cmdlets.
    /// </remarks>
    private static void Main(string[] args)
    {
        Collection<PSObject> result;      // Will hold the result
                                         // of running the cmdlets.

        // Create a runspace. We can't use the RunspaceInvoke class
        // because we need to get at the underlying runspace to
        // explicitly add the commands. Notice that no PSHost object is
        // supplied to the CreateRunspace method so the default host is
        // used. See the Host samples for more information on creating
        // your own custom host.
        using (Runspace myRunSpace = RunspaceFactory.CreateRunspace())
        {
            myRunSpace.Open();

            // Create a PowerShell object and specify the runspace.
            PowerShell powershell = PowerShell.Create();
            powershell.Runspace = myRunSpace;

            // Use the using statement so we dispose of the PowerShell object
            // when we're done.
            using (powershell)
            {
                // Add the Get-Process cmdlet to the PowerShell object. Notice
                // we are specify the name of the cmdlet, not a script.
                powershell.AddCommand("Get-Process");

                // Add the Measure-Object cmdlet to count the number
                // of objects being returned. Commands are always added to the end
                // of the pipeline.
                powershell.AddCommand("Measure-Object");

                // Run the cmdlets synchronously and save the objects returned.
            }
        }
    }
}
```

```

        result = powershell.Invoke();
    }

    // Even after disposing of the pipeLine, we still need to set
    // the powershell variable to null so that the garbage collector
    // can clean it up.
    powershell = null;

    // Display the results of running the commands (checking that
    // everything is ok first.
    if (result == null || result.Count != 1)
    {
        throw new InvalidOperationException(
            "pipeline.Invoke() returned the wrong number of
objects");
    }

    PSMemberInfo count = result[0].Properties["Count"];
    if (count == null)
    {
        throw new InvalidOperationException(
            "The object returned doesn't have a 'count' property");
    }

    Console.WriteLine(
        "Runspace07: The Get-Process cmdlet returned {0}
objects",
        count.Value);

    // Close the runspace to release any resources.
    myRunSpace.Close();
}

System.Console.WriteLine("Hit any key to exit...");
System.Console.ReadKey();
}
}
}

```

See Also

[Writing a Windows PowerShell Host Application](#)

Runspace08 Sample

Article • 03/24/2025

This sample shows how to add commands and arguments to the pipeline of a [System.Management.Automation.PowerShell](#) object and how to run the commands synchronously.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Creating a [System.Management.Automation.Runspaces.Runspace](#) object by using the [System.Management.Automation.Runspaces.RunspaceFactory](#) class.
- Creating a [System.Management.Automation.PowerShell](#) object that uses the runspace.
- Adding cmdlets to the pipeline of the [System.Management.Automation.PowerShell](#) object.
- Running the cmdlets synchronously.
- Extracting properties from the [System.Management.Automation.PSObject](#) objects returned by the command.

Example

This sample runs the [Get-Process](#) and [Sort-Object](#) cmdlets by using a [System.Management.Automation.PowerShell](#) object.

```
C#
```

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
```

```

using PowerShell = System.Management.Automation.PowerShell;

/// <summary>
/// This class contains the Main entry point for this host application.
/// </summary>
internal class Runspace08
{
    /// <summary>
    /// This sample shows how to use a PowerShell object to run commands.
    The
        /// PowerShell object builds a pipeline that include the Get-Process
        cmdlet,
        /// which is then piped to the Sort-Object cmdlet. Parameters are added
        to the
            /// Sort-Object cmdlet to sort the HandleCount property in descending
            order.
        /// </summary>
        /// <param name="args">Parameter is not used.</param>
        /// <remarks>
        /// This sample demonstrates:
        /// 1. Creating a PowerShell object
        /// 2. Adding individual commands to the PowerShell object.
        /// 3. Adding parameters to the commands.
        /// 4. Running the pipeline of the PowerShell object synchronously.
        /// 5. Working with PSObject objects to extract properties
        ///     from the objects returned by the commands.
        /// </remarks>
    private static void Main(string[] args)
    {
        Collection<PSObject> results; // Holds the result of the pipeline
        execution.

            // Create the PowerShell object. Notice that no runspace is specified
            so a
            // new default runspace is used.
            PowerShell powershell = PowerShell.Create();

            // Use the using statement so that we can dispose of the PowerShell
            object
            // when we are done.
            using (powershell)
            {
                // Add the Get-Process cmdlet to the pipeline of the PowerShell
                object.
                powershell.AddCommand("Get-Process");

                // Add the Sort-Object cmdlet and its parameters to the pipeline of
                // the PowerShell object so that we can sort the HandleCount
                property
                    // in descending order.
                    powershell.AddCommand("Sort-
                Object").AddParameter("Descending").AddParameter("Property", "HandleCount");

                    // Run the commands of the pipeline synchronously.
                    results = powershell.Invoke();
    }
}

```

```
}

// Even after disposing of the PowerShell object, we still
// need to set the powershell variable to null so that the
// garbage collector can clean it up.
powershell = null;

Console.WriteLine("Process           HandleCount");
Console.WriteLine("-----");

// Display the results returned by the commands.
foreach (PSObject result in results)
{
    Console.WriteLine(
        "{0,-20} {1}",
        result.Members["ProcessName"].Value,
        result.Members["HandleCount"].Value);
}

System.Console.WriteLine("Hit any key to exit...");
System.Console.ReadKey();
}
}
```

See Also

[Writing a Windows PowerShell Host Application](#)

Runspace09 Sample

Article • 03/24/2025

This sample shows how to add a script to the pipeline of a [System.Management.Automation.PowerShell](#) object and how to run the script asynchronously. Events are used to handle the output of the script.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Creating a [System.Management.Automation.PowerShell](#) object that uses the runspace.
- Adding a script the pipeline of the [System.Management.Automation.PowerShell](#) object.
- Using the [System.Management.Automation.PowerShell.BeginInvoke*](#) method to run the pipeline asynchronously.
- Using the events of the [System.Management.Automation.PowerShell](#) object to process the output of the script.
- Using the [System.Management.Automation.PowerShell.Stop*](#) method to interrupt the invocation of the pipeline.

Example

This sample runs to run a script that generates the numbers from 1 to 10 with delays between each number. The script is run asynchronously and events are used to handle the output.

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.Generic;
```

```

using System.Collections.ObjectModel;
using System.Diagnostics;
using System.Management.Automation;
using System.Management.Automation.Runspaces;
using PowerShell = System.Management.Automation.PowerShell;

/// <summary>
/// This class contains the Main entry point for this host application.
/// </summary>
internal class Runspace09
{
    /// <summary>
    /// This sample shows how to use a PowerShell object to run a
    /// script that generates the numbers from 1 to 10 with delays
    /// between each number. The pipeline of the PowerShell object
    /// is run asynchronously and events are used to handle the output.
    /// </summary>
    /// <param name="args">The parameter is not used.</param>
    /// <remarks>
    /// This sample demonstrates the following:
    /// 1. Creating a PowerShell object.
    /// 2. Adding a script to the pipeline of the PowerShell object.
    /// 3. Using the BeginInvoke method to run the pipeline asynchronously.
    /// 4. Using the events of the PowerShell object to process the
    ///     output of the script.
    /// 5. Using the PowerShell.Stop() method to interrupt the invocation of
    ///     the pipeline.
    /// </remarks>
    private static void Main(string[] args)
    {
        Console.WriteLine("Print the numbers from 1 to 10. Hit any key to halt
processing\n");

        using (PowerShell powershell = PowerShell.Create())
        {
            // Add a script to the PowerShell object. The script generates the
            // numbers from 1 to 10 in half second intervals.
            powershell.AddScript("1..10 | foreach {$_. ; Start-Sleep -Milli
500}");

            // Add the event handlers. If we did not care about hooking the
            DataAdded
                // event, we would let BeginInvoke create the output stream for us.
                PSDataCollection<PSObject> output = new PSDataCollection<PSObject>
();
                output.DataAdded += new EventHandler<DataAddedEventArgs>
(Output_DataAdded);
                powershell.InvocationStateChanged += new
EventHandler<PSInvocationStateChangedEventArgs>
(Powershell_InvocationStateChanged);

            // Invoke the pipeline asynchronously.
            IAsyncResult asyncResult = powershell.BeginInvoke<PSObject,
PSObject>(null, output);

```

```

        // Wait for things to happen. If the user hits a key before the
        // script has completed, then call the PowerShell Stop() method
        // to halt processing.
        Console.ReadKey();
        if (powershell.InvocationStateInfo.State !=
PSInvocationState.Completed)
        {
            // Stop the invocation of the pipeline.
            Console.WriteLine("\nStopping the pipeline!\n");
            powershell.Stop();

            // Wait for the Windows PowerShell state change messages to be
            displayed.
            System.Threading.Thread.Sleep(500);
            Console.WriteLine("\nPress a key to exit");
            Console.ReadKey();
        }
    }
}

/// <summary>
/// The output data added event handler. This event is called when
/// data is added to the output pipe. It reads the data that is
/// available and displays it on the console.
/// </summary>
/// <param name="sender">The output pipe this event is associated with.
</param>
/// <param name="e">Parameter is not used.</param>
private static void Output_DataAdded(object sender, DataAddedEventArgs
e)
{
    PSDataCollection<PSObject> myp = (PSDataCollection<PSObject>)sender;

    Collection<PSObject> results = myp.ReadAll();
    foreach (PSObject result in results)
    {
        Console.WriteLine(result.ToString());
    }
}

/// <summary>
/// This event handler is called when the pipeline state is changed.
/// If the state change is to Completed, the handler issues a message
/// asking the user to exit the program.
/// </summary>
/// <param name="sender">This parameter is not used.</param>
/// <param name="e">The PowerShell state information.</param>
private static void Powershell_InvocationStateChanged(object sender,
PSInvocationStateChangedEventArgs e)
{
    Console.WriteLine("PowerShell object state changed: state: {0}\n",
e.InvocationStateInfo.State);
    if (e.InvocationStateInfo.State == PSInvocationState.Completed)
    {
        Console.WriteLine("Processing completed, press a key to exit!");
    }
}

```

```
        }  
    }  
}
```

See Also

[Writing a Windows PowerShell Host Application](#)

Runspace10 Sample

Article • 03/24/2025

This sample shows how to create a default initial session state, how to add a cmdlet to the [System.Management.Automation.Runspaces.InitialSessionState](#), how to create a runspace that uses the initial session state, and how to run the command by using a [System.Management.Automation.PowerShell](#) object.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Creating a [System.Management.Automation.Runspaces.InitialSessionState](#) object.
- Adding a cmdlet (defined by the Host application) to the [System.Management.Automation.Runspaces.InitialSessionState](#) object.
- Creating a [System.Management.Automation.Runspaces.Runspace](#) object that uses the object.
- Creating a [System.Management.Automation.PowerShell](#) object that uses the [System.Management.Automation.Runspaces.Runspace](#) object.
- Adding the command to the pipeline of the [System.Management.Automation.PowerShell](#) object.
- Extracting properties from the [System.Management.Automation.PSObject](#) objects returned by the command.

Example

This sample creates a runspace that uses a [System.Management.Automation.Runspaces.InitialSessionState](#) object to define the elements that are available when the runspace is opened. In this sample, the Get-Proc cmdlet (defined by the Host application) is added to the initial session state, and the cmdlet is run synchronously by using a [System.Management.Automation.PowerShell](#) object.

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Diagnostics;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    #region GetProcCommand

    /// <summary>
    /// Class that implements the GetProcCommand.
    /// </summary>
    [Cmdlet(VerbsCommon.Get, "Proc")]
    public class GetProcCommand : Cmdlet
    {
        #region Cmdlet Overrides

        /// <summary>
        /// For each of the requested process names, retrieve and write
        /// the associated processes.
        /// </summary>
        protected override void ProcessRecord()
        {
            // Get the current processes.
            Process[] processes = Process.GetProcesses();

            // Write the processes to the pipeline making them available
            // to the next cmdlet. The second argument (true) tells the
            // system to enumerate the array, and send one process object
            // at a time to the pipeline.
            WriteObject(processes, true);
        }

        #endregion Overrides
    } // End GetProcCommand class.

    #endregion GetProcCommand

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace10
    {
        /// <summary>
        /// This sample shows how to create a default initial session state, how
        /// to add
        /// add a cmdlet to the InitialSessionState object, and then how to
        /// create
        /// a Runspace object.
    }
}
```

```
/// </summary>
/// <param name="args">Parameter is not used.</param>
/// This sample demonstrates:
/// 1. Creating an InitialSessionState object.
/// 2. Adding a cmdlet to the InitialSessionState object.
/// 3. Creating a runspace that uses the InitialSessionState object.
/// 4. Creating a PowerShell object that uses the Runspace object.
/// 5. Running the added command synchronously.
/// 6. Working with PSObject objects to extract properties
///     from the objects returned by the pipeline.
private static void Main(string[] args)
{
    // Create a default InitialSessionState object. The default
    // InitialSessionState object contains all the elements provided
    // by Windows PowerShell.
    InitialSessionState iss = InitialSessionState.CreateDefault();

    // Add the Get-Proc cmdlet to the InitialSessionState object.
    SessionStateCmdletEntry ssce = new SessionStateCmdletEntry("Get-Proc",
typeof(GetProcCommand), null);
    iss.Commands.Add(ssce);

    // Create a Runspace object that uses the InitialSessionState object.
    // Notice that no PSHost object is specified, so the default host is
used.
    // See the Hosting samples for information on creating your own custom
host.
    using (Runspace myRunSpace = RunspaceFactory.CreateRunspace(iss))
    {
        myRunSpace.Open();

        using (PowerShell powershell = PowerShell.Create())
        {
            powershell.Runspace = myRunSpace;

            // Add the Get-Proc cmdlet to the pipeline of the PowerShell
object.
            powershell.AddCommand("Get-Proc");

            Collection<PSObject> results = powershell.Invoke();

            Console.WriteLine("Process           HandleCount");
            Console.WriteLine("-----");

            // Display the output of the pipeline.
            foreach (PSObject result in results)
            {
                Console.WriteLine(
                    "{0,-20} {1}",
                    result.Members["ProcessName"].Value,
                    result.Members["HandleCount"].Value);
            }
        }

        // Close the runspace to release resources.
    }
}
```

```
    myRunSpace.Close();
}

System.Console.WriteLine("Hit any key to exit...");
System.Console.ReadKey();
}
}
```

See Also

[Writing a Windows PowerShell Host Application](#)

Runspace11 Sample

Article • 03/24/2025

This sample shows how to use the [System.Management.Automation.ProxyCommand](#) class to create a proxy command that calls an existing cmdlet, but restricts the set of available parameters. The proxy command is then added to an initial session state that is used to create a constrained runspace. This means that the user can access the functionality of the cmdlet only through the proxy command.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

This sample demonstrates the following.

- Creating a [System.Management.Automation.CommandMetadata](#) object that describes the metadata of an existing cmdlet.
- Creating a [System.Management.Automation.Runspaces.InitialSessionState](#) object.
- Modifying the cmdlet metadata to remove a parameter of the cmdlet.
- Adding the cmdlet to the [System.Management.Automation.Runspaces.InitialSessionState](#) object and making the cmdlet private.
- Creating a proxy function that calls the existing cmdlet, but exposes only a restricted set of parameters.
- Adding the proxy function to the initial session state.
- Creating a [System.Management.Automation.PowerShell](#) object that uses the [System.Management.Automation.Runspaces.Runspace](#) object.
- Calling the private cmdlet and the proxy function using a [System.Management.Automation.PowerShell](#) object to demonstrate the constrained runspace.

Example

This creates a proxy command for a private cmdlet to demonstrate a constrained runspace.

```
C#  
  
namespace Microsoft.Samples.PowerShell.Runspaces  
{  
    using System;  
    using System.Collections.Generic;  
    using System.Diagnostics;  
    using System.Management.Automation;  
    using System.Management.Automation.Runspaces;  
    using PowerShell = System.Management.Automation.PowerShell;  
  
    #region GetProcCommand  
  
    /// <summary>  
    /// This class implements the Get-Proc cmdlet. It has been copied  
    /// verbatim from the GetProcessSample02.cs sample.  
    /// </summary>  
    [Cmdlet(VerbsCommon.Get, "Proc")]  
    public class GetProcCommand : Cmdlet  
    {  
        #region Parameters  
  
        /// <summary>  
        /// The names of the processes to act on.  
        /// </summary>  
        private string[] processNames;  
  
        /// <summary>  
        /// Gets or sets the list of process names on which  
        /// the Get-Proc cmdlet will work.  
        /// </summary>  
        [Parameter(Position = 0)]  
        [ValidateNotNullOrEmpty]  
        public string[] Name  
        {  
            get { return this.processNames; }  
            set { this.processNames = value; }  
        }  
  
        #endregion Parameters  
  
        #region Cmdlet Overrides  
  
        /// <summary>  
        /// The ProcessRecord method calls the Process.GetProcesses  
        /// method to retrieve the processes specified by the Name  
        /// parameter. Then, the WriteObject method writes the  
        /// associated processes to the pipeline.  
        /// </summary>  
        protected override void ProcessRecord()  
        {
```

```

        // If no process names are passed to the cmdlet, get all
        // processes.
        if (this.processNames == null)
        {
            WriteObject(Process.GetProcesses(), true);
        }
        else
        {
            // If process names are passed to cmdlet, get and write
            // the associated processes.
            foreach (string name in this.processNames)
            {
                WriteObject(Process.GetProcessesByName(name), true);
            }
        } // if (processNames...
    } // ProcessRecord

    #endregion Cmdlet Overrides
} // GetProcCommand

#endregion GetProcCommand

/// <summary>
/// This class contains the Main entry point for this host application.
/// </summary>
internal class Runspace11
{
    /// <summary>
    /// This shows how to use the ProxyCommand class to create a proxy
    /// command that calls an existing cmdlet, but restricts the set of
    /// available parameters. The proxy command is then added to an initial
    /// session state that is used to create a constrained runspace. This
    /// means that the user can access the cmdlet only through the proxy
    /// command.
    /// </summary>
    /// <remarks>
    /// This sample demonstrates the following:
    /// 1. Creating a CommandMetadata object that describes the metadata of
    an
    ///     existing cmdlet.
    /// 2. Modifying the cmdlet metadata to remove a parameter of the
    cmdlet.
    /// 3. Adding the cmdlet to an initial session state and making it
    private.
    /// 4. Creating a proxy function that calls the existing cmdlet, but
    exposes
    ///     only a restricted set of parameters.
    /// 6. Adding the proxy function to the initial session state.
    /// 7. Calling the private cmdlet and the proxy function to demonstrate
    the
    ///     constrained runspace.
    /// </remarks>
    private static void Main()
    {
        // Create a default initial session state. The default initial session

```

```

state
    // includes all the elements that are provided by Windows PowerShell.
InitialSessionState iss = InitialSessionState.CreateDefault();

    // Add the Get-Proc cmdlet to the initial session state.
    SessionStateCmdletEntry cmdletEntry = new
SessionStateCmdletEntry("Get-Proc", typeof(GetProcCommand), null);
    iss.Commands.Add(cmdletEntry);

    // Make the cmdlet private so that it is not accessible.
    cmdletEntry.Visibility = SessionStateEntryVisibility.Private;

    // Set the language mode of the initial session state to NoLanguage to
    // prevent users from using language features. Only the invocation of
    // public commands is allowed.
    iss.LanguageMode = PSLanguageMode.NoLanguage;

    // Create the proxy command using cmdlet metadata to expose the
    // Get-Proc cmdlet.
    CommandMetadata cmdletMetadata = new
CommandMetadata(typeof(GetProcCommand));

    // Remove one of the parameters from the command metadata.
    cmdletMetadata.Parameters.Remove("Name");

    // Generate the body of a proxy function that calls the original
cmdlet,
    // but does not have the removed parameter.
    string bodyOfProxyFunction = ProxyCommand.Create(cmdletMetadata);

    // Add the proxy function to the initial session state. The name of
the proxy
    // function can be the same as the name of the cmdlet, but to clearly
    // demonstrate that the original cmdlet is not available a different
name is
    // used for the proxy function.
    iss.Commands.Add(new SessionStateFunctionEntry("Get-ProcProxy",
bodyOfProxyFunction));

    // Create the constrained runspace using the initial session state.
using (Runspace myRunspace = RunspaceFactory.CreateRunspace(iss))
{
    myRunspace.Open();

    // Call the private cmdlet to demonstrate that it is not available.
    try
    {
        using (PowerShell powershell = PowerShell.Create())
        {
            powershell.Runspace = myRunspace;
            powershell.AddCommand("Get-Proc").AddParameter("Name",
"**explore**");
            powershell.Invoke();
        }
    }
}

```

```

        catch (CommandNotFoundException e)
    {
        System.Console.WriteLine(
            "Invoking 'Get-Proc' failed as expected: {0}: {1}",
            e.GetType().FullName,
            e.Message);
    }

    // Call the proxy function to demonstrate that the -Name parameter
is
    // not available.
    try
    {
        using (PowerShell powershell = PowerShell.Create())
        {
            powershell.Runspace = myRunspace;
            powershell.AddCommand("Get-ProcProxy").AddParameter("Name",
"idle");
            powershell.Invoke();
        }
    }
    catch (ParameterBindingException e)
    {
        System.Console.WriteLine(
            "\nInvoking 'Get-ProcProxy -Name idle' failed as
expected: {0}: {1}",
            e.GetType().FullName,
            e.Message);
    }

    // Call the proxy function to demonstrate that it calls into the
    // private cmdlet to retrieve the processes.
    using (PowerShell powershell = PowerShell.Create())
    {
        powershell.Runspace = myRunspace;
        powershell.AddCommand("Get-ProcProxy");
        List<Process> processes = new List<Process>
(powershell.Invoke<Process>());
        System.Console.WriteLine(
            "\nInvoking the Get-ProcProxy function called into
the Get-Proc cmdlet and returned {0} processes",
            processes.Count);
    }

    // Close the runspace to release resources.
    myRunspace.Close();
}

System.Console.WriteLine("Hit any key to exit...");
System.Console.ReadKey();
}
}

```

See Also

[Writing a Windows PowerShell Host Application](#)

Remote Runspace Samples

Article • 09/17/2021

This section includes sample code that shows how to create runspaces that can be used to connect to a computer by using WS-Management-based Windows PowerShell remoting. You can use Microsoft Visual Studio to create a console application and then copy the code from the topics in this section into your host application.

In This Section

Note

For more information about running commands on a remote computer, see [Windows PowerShell Remoting](#).

[RemoteRunspace01 Sample](#) This sample shows how to create a remote runspace that is used to establish a remote connection.

[RemoteRunspacePool01 Sample](#) This sample shows how to construct a remote runspace pool and how to run multiple commands concurrently by using this pool.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

RemoteRunspace01 Sample

Article • 03/24/2025

This sample shows how to create a remote runspace that is used to establish a remote connection.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

- Creating a [System.Management.Automation.Runspaces.WSManConnectionInfo](#) object.
- Setting the [System.Management.Automation.Runspaces.RunspaceConnectionInfo.OperationTimeout*](#) and [System.Management.Automation.Runspaces.RunspaceConnectionInfo.OpenTimeout*](#) properties of the [System.Management.Automation.Runspaces.WSManConnectionInfo](#) object.
- Creating a remote runspace that uses the [System.Management.Automation.Runspaces.WSManConnectionInfo](#) object to establish the remote connection.
- Closing the remote runspace to release the remote connection.

Example

This sample defines a remote connection and then uses that connection information to establish a remote connection.

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Management.Automation; // Windows PowerShell
    namespace
    {
        using System.Management.Automation.Runspaces; // Windows PowerShell
        namespace.
```

```

/// <summary>
/// This class contains the Main entry point for the application.
/// </summary>
internal class RemoteRunspace01
{
    /// <summary>
    /// This sample shows how to use a WSMANConnectionInfo object to set
    /// various timeouts and how to establish a remote connection.
    /// </summary>
    /// <param name="args">This parameter is not used.</param>
    public static void Main(string[] args)
    {
        // Create a WSMANConnectionInfo object using the default constructor
        // to connect to the "localhost". The WSMANConnectionInfo object can
        // also specify connections to remote computers.
        WSMANConnectionInfo connectionInfo = new WSMANConnectionInfo();

        // Set the OperationTimeout property. The OperationTimeout is used to
        tell
        // Windows PowerShell how long to wait (in milliseconds) before timing
        out
        // for any operation. This includes sending input data to the remote
        computer,
        // receiving output data from the remote computer, and more. The user
        can
        // change this timeout depending on whether the connection is to a
        computer
        // in the data center or across a slow WAN.
        connectionInfo.OperationTimeout = 4 * 60 * 1000; // 4 minutes.

        // Set the OpenTimeout property. OpenTimeout is used to tell Windows
        PowerShell
        // how long to wait (in milliseconds) before timing out while
        establishing a
        // remote connection. The user can change this timeout depending on
        whether the
        // connection is to a computer in the data center or across a slow
        WAN.
        connectionInfo.OpenTimeout = 1 * 60 * 1000; // 1 minute.

        // Create a remote runspace using the connection information.
        using (Runspace remoteRunspace =
RunspaceFactory.CreateRunspace(connectionInfo))
        {
            // Establish the connection by calling the Open() method to open the
            runspace.
            // The OpenTimeout value set previously will be applied while
            establishing
            // the connection. Establishing a remote connection involves sending
            and
            // receiving some data, so the OperationTimeout will also play a
            role in this process.
            remoteRunspace.Open();

```

```
// Add the code to run commands in the remote runspace here. The
// OperationTimeout value set previously will play a role here
because
    // running commands involves sending and receiving data.

    // Close the connection. Call the Close() method to close the remote
    // runspace. The Dispose() method (called by using primitive) will
call
    // the Close() method if it is not already called.
    remoteRunspace.Close();
}
}
}
}
```

RemoteRunspacePool01 Sample

Article • 03/24/2025

This sample shows how to construct a remote runspace pool and how to run multiple commands concurrently by using this pool.

Requirements

This sample requires Windows PowerShell 2.0.

Demonstrates

- Creating a [System.Management.Automation.Runspaces.WSManConnectionInfo](#) object.
- Setting the [System.Management.Automation.Runspaces.RunspaceConnectionInfo.OperationTimeout*](#) and [System.Management.Automation.Runspaces.RunspaceConnectionInfo.OpenTimeout*](#) properties of the [System.Management.Automation.Runspaces.WSManConnectionInfo](#) object.
- Creating a remote runspace that uses the [System.Management.Automation.Runspaces.WSManConnectionInfo](#) object to establish the remote connection.
- Running the [Get-Process](#) and [Get-Service](#) cmdlets concurrently by using the remote runspace pool.
- Closing the remote runspace pool to release the remote connection.

Example

This sample shows how to construct a remote runspace pool and how to run multiple commands concurrently by using this pool.

C#

```
namespace Samples
{
    using System;
    using System.Management.Automation;           // Windows PowerShell
```

```
namespace.
using System.Management.Automation.Runspaces; // Windows PowerShell
namespace.

/// <summary>
/// This class contains the Main entry point for the application.
/// </summary>
internal class RemoteRunspacePool01
{
    /// <summary>
    /// This sample shows how to construct a remote RunspacePool and how to
    /// concurrently run the Get-Process and Get-Service commands using the
    /// runspaces of the pool.
    /// </summary>
    /// <param name="args">Parameter is not used.</param>
    public static void Main(string[] args)
    {
        // Create a WSMANConnectionInfo object using the default constructor
        to
        // connect to the "localhost". The WSMANConnectionInfo object can also
        // specify connections to remote computers.
        WSMANConnectionInfo connectionInfo = new WSMANConnectionInfo();

        // Create a remote runspace pool that uses the WSMANConnectionInfo
        object.
        // The minimum runspaces value of 1 specifies that Windows PowerShell
        will
        // keep at least 1 runspace open. The maximum runspaces value of 2
        specifies
        // that Windows PowerShell will allow 2 runspaces to be opened at the
        // same time so that two commands can be run concurrently.
        using (RunspacePool remoteRunspacePool =
            RunspaceFactory.CreateRunspacePool(1, 2, connectionInfo))
        {
            // Call the Open() method to open the runspace pool and establish
            // the connection.
            remoteRunspacePool.Open();

            // Call the Create() method to create a pipeline, call the
            AddCommand(string)
                // method to add the "Get-Process" command, and then call the
            BeginInvoke()
                // method to run the command asynchronously using a runspace of the
            pool.
            PowerShell gpsCommand = PowerShell.Create().AddCommand("Get-
            Process");
            gpsCommand.RunspacePool = remoteRunspacePool;
            IAsyncResult gpsCommandAsyncResult = gpsCommand.BeginInvoke();

            // The previous call does not block the current thread because it is
            // running asynchronously. Because the remote runspace pool can open
            two
            // runspace, the second command can be run.
            PowerShell getServiceCommand = PowerShell.Create().AddCommand("Get-
            Service");
        }
    }
}
```

```

getServiceCommand.RunspacePool = remoteRunspacePool;
IAsyncResult getServiceCommandAsyncResult =
getServiceCommand.BeginInvoke();

    // When you are ready to handle the output, wait for the command to
    // complete
    // before extracting results. A call to the EndInvoke() method will
    // block and return
    // the output.
    PSDataCollection<PSObject> gpsCommandOutput =
gpsCommand.EndInvoke(gpsCommandAsyncResult);

    // Process the output from the first command.
    if ((gpsCommandOutput != null) && (gpsCommandOutput.Count > 0))
{
    Console.WriteLine("The first output from running Get-Process
command: ");
    Console.WriteLine(
        "Process Name: {0} Process Id: {1}",
        gpsCommandOutput[0].Properties["ProcessName"].Value,
        gpsCommandOutput[0].Properties["Id"].Value);
    Console.WriteLine();
}

    // Now process the output from the second command. As discussed
previously, wait
    // for the command to complete before extracting the results.
    PSDataCollection<PSObject> getServiceCommandOutput =
getServiceCommand.EndInvoke(
    getServiceCommandAsyncResult);

    // Process the output of the second command as needed.
    if ((getServiceCommandOutput != null) &&
(getServiceCommandOutput.Count > 0))
{
    Console.WriteLine("The first output from running Get-Service
command: ");
    Console.WriteLine(
        "Service Name: {0} Description: {1} State: {2}",
        getServiceCommandOutput[0].Properties["ServiceName"].Value,
        getServiceCommandOutput[0].Properties["DisplayName"].Value,
        getServiceCommandOutput[0].Properties["Status"].Value);
}

    // Once done with running all the commands, close the remote
runspace pool.
    // The Dispose() method (called by using primitive) will call
Close(), if it
    // is not already called.
    remoteRunspacePool.Close();
} // End Using.

```

```
    } // End Main.  
} // End RemoteRunspacePool01 class  
}
```

See Also

Formatting File Overview

Article • 03/13/2023

The display format for the objects that are returned by commands (cmdlets, functions, and scripts) are defined by using formatting files (`format.ps1xml`). Several of these files are provided by PowerShell to define the display format for those objects returned by PowerShell-provided commands, such as the `System.Diagnostics.Process` object returned by the `Get-Process` cmdlet. However, you can also create your own custom formatting files to overwrite the default display formats or you can write a custom formatting file to define the display of objects returned by your own commands.

Important

Formatting files do not determine the elements of an object that are returned to the pipeline. When an object is returned to the pipeline, all members of that object are available even if some are not displayed.

PowerShell uses the data in these formatting files to determine what is displayed and how the displayed data is formatted. The displayed data can include the properties of an object or the value of a script. Scripts are used if you want to display some value that is not available directly from the properties of an object, such as adding the value of two properties of an object and then displaying the sum as a piece of data. Formatting of the displayed data is done by defining views for the objects that you want to display. You can define a single view for each object, you can define a single view for multiple objects, or you can define multiple views for the same object. There is no limit to the number of views that you can define.

Common Features of Formatting Files

Each formatting file can define the following components that can be shared across all the views defined by the file:

- Default configuration setting, such as whether the data displayed in the rows of tables will be displayed on the next line if the data is longer than the width of the column. For more information about these settings, see [Wrap Element for TableRowEntry](#).
- Sets of objects that can be displayed by any of the views of the formatting file. For more information about these sets (referred to as **selection sets**), see [Defining Sets of Objects](#).

- Common controls that can be used by all the views of the formatting file. Controls give you finer control on how data is displayed. For more information about controls, see [Defining Custom Controls](#).

Formatting Views

Formatting views can display objects in a table format, list format, wide format, and custom format. For the most part, each formatting definition is described by a set of XML tags that describe the view. Each view contains the name of the view, the objects that use the view, and the elements of the view, such as the column and row information for a table view.

Table View

Lists the properties of an object or a script block value in one or more columns. Each column represents a single property of the object or a script value. You can define a table view that displays all the properties of an object, a subset of the properties of an object, or a combination of properties and script values. Each row of the table represents a returned object. Creating a table view is very similar to when you pipe an object to the `Format-Table` cmdlet. For more information about this view, see [Table View](#).

List View

Lists the properties of an object or a script value in a single column. Each row of the list displays an optional label or the property name followed by the value of the property or script. Creating a list view is very similar to piping an object to the `Format-List` cmdlet. For more information about this view, see [List View](#).

Wide View

Lists a single property of an object or a script value in one or more columns. There is no label or header for this view. Creating a wide view is very similar to piping an object to the `Format-Wide` cmdlet. For more information about this view, see [Wide View](#).

Custom View

Displays a customizable view of object properties or script values that does not adhere to the rigid structure of table views, list views, or wide views. You can define a stand-alone custom view, or you can define a custom view that is used by another view, such

as a table view or list view. Creating a custom view is very similar to piping an object to the `Format-Custom` cmdlet. For more information about this view, see [Custom View](#).

Components of a View

The following XML examples show the basic XML components of a view. The individual XML elements vary depending on which view you want to create, but the basic components of the views are all the same.

To start with, each view has a `Name` element that specifies a user friendly name that is used to reference the view. a `ViewSelectedBy` element that defines which .NET objects are displayed by the view, and a `control` element that defines the view.

XML

```
<ViewDefinitions>
  <View>
    <Name>NameOfView</Name>
    <ViewSelectedBy>...</ViewSelectedBy>
    <TableControl>...</TableControl>
  </View>
  <View>
    <Name>NameOfView</Name>
    <ViewSelectedBy>...</ViewSelectedBy>
    <ListControl>...</ListControl>
  </View>
  <View>
    <Name>NameOfView</Name>
    <ViewSelectedBy>...</ViewSelectedBy>
    <WideControl>...</WideControl>
  </View>
  <View>
    <Name>NameOfView</Name>
    <ViewSelectedBy>...</ViewSelectedBy>
    <CustomControl>...</CustomControl>
  </View>
</ViewDefinitions>
```

Within the control element, you can define one or more `entry` elements. If you use multiple definitions, you must specify which .NET objects use each definition. Typically only one entry, with only one definition, is needed for each control.

XML

```
<ListControl>
  <ListEntries>
    <ListEntry>
```

```
<EntrySelectedBy>...</EntrySelectedBy>
<ListItems>...</ListItems>
</ListEntry>
<ListEntry>
    <EntrySelectedBy>...</EntrySelectedBy>
    <ListItems>...</ListItems>
</ListEntry>
<ListEntry>
    <EntrySelectedBy>...</EntrySelectedBy>
    <ListItems>...</ListItems>
</ListEntry>
</ListEntries>
</ListControl>
```

Within each entry element of a view, you specify the **item** elements that define the .NET properties or scripts that are displayed by that view.

XML

```
<ListItems>
    <ListItem>...</ListItem>
    <ListItem>...</ListItem>
    <ListItem>...</ListItem>
</ListItems>
```

As shown in the preceding examples, the formatting file can contain multiple views, a view can contain multiple definitions, and each definition can contain multiple items.

Example of a Table View

The following example shows the XML tags used to define a table view that contains two columns. The **ViewDefinitions** element is the container element for all the views defined in the formatting file. The **View** element defines the specific table, list, wide, or custom view. Within each **View** element, the **Name** element specifies the name of the view, the **ViewSelectedBy** element defines the objects that use the view, and the different control elements (such as the **TableControl** element shown in the following example) define the type of the view.

XML

```
<ViewDefinitions>
    <View>
        <Name>Name of View</Name>
        <ViewSelectedBy>
            <TypeName>Object to display using this view</TypeName>
```

```
<TypeName>Object to display using this view</TypeName>
</ViewSelectedBy>
<TableControl>
  <TableHeaders>
    <TableColumnHeader>
      <Width></Width>
    </TableColumnHeader>
    <TableColumnHeader>
      <Width></Width>
    </TableColumnHeader>
  </TableHeaders>
  <TableRowEntries>
    <TableRowEntry>
      <TableColumnItems>
        <TableColumnItem>
          <PropertyName>Header for column 1</PropertyName>
        </TableColumnItem>
        <TableColumnItem>
          <PropertyName>Header for column 2</PropertyName>
        </TableColumnItem>
      </TableColumnItems>
    </TableRowEntry>
  </TableRowEntries>
</TableControl>
</View>
</ViewDefinitions>
```

See Also

[Creating a List View](#)

[Creating a Table View](#)

[Creating a Wide View](#)

[Creating Custom Controls](#)

[Writing a PowerShell Formatting and Types File](#)

Formatting File Concepts

Article • 09/17/2021

The topics in this section provide information that you might need to know when creating your own formatting files, such as the different types of views that you can define and the special components of those views.

In This Section

[Creating a Table View](#) Provides an example of a displayed table view and the XML elements used to define the view.

[Creating a List View](#) Provides an example of a displayed list view and the XML elements used to define the view.

[Creating a Wide View](#) Provides an example of a displayed wide view and the XML elements used to define the view.

[Creating Custom Controls](#) Provides an example of a custom control.

[Defining Selection Sets](#) Provides information, an example, and describes the XML elements used to create a selection set.

[Defining Conditions for Displaying Data](#) When defining what data is displayed by a view or a control, you can specify a condition that must exist for the data to be displayed.

[Formatting Displayed Data](#) You can specify how the individual data points in your List, Table, or Wide view are displayed.

[PowerShell Formatting Files](#) Lists the available formatting files provided by PowerShell.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Creating a Table View

Article • 01/20/2022

A table view displays data in one or more columns. Each row in the table represents a .NET object, and each column of the table represents a property of the object or a script value. You can define a table view that displays all the properties of an object or a subset of the properties of an object.

A Table View Display

The following example shows how Windows PowerShell displays the [System.ServiceProcess.ServiceController](#) object that is returned by the [Get-Service](#) cmdlet. For this object, Windows PowerShell has defined a table view that displays the `Status` property, the `Name` property (this property is an alias property for the `ServiceName` property), and the `DisplayName` property. Each row in the table represents an object returned by the cmdlet.

Output		
Status	Name	DisplayName
Stopped	AJRouter	AllJoyn Router Service
Stopped	ALG	Application Layer Gateway Service
Stopped	AppIDSvc	Application Identity
Running	Appinfo	Application Information

Defining the Table View

The following XML shows the table view schema for displaying the [System.ServiceProcess.ServiceController](#) object. You must specify each property that you want displayed in the table view.

```
XML
<View>
  <Name>service</Name>
  <ViewSelectedBy>
    <TypeName>System.ServiceProcess.ServiceController</TypeName>
  </ViewSelectedBy>
  <TableControl>
    <TableHeaders>
      < TableColumnHeader>
        <Width>8</Width>
```

```

</TableColumnHeader>
<TableColumnHeader>
  <Width>18</Width>
</TableColumnHeader>
<TableColumnHeader>
  <Width>38</Width>
</TableColumnHeader>
</TableHeaders>
<TableRowEntries>
  <TableRowEntry>
    <TableColumnItems>
      <TableColumnItem>
        <PropertyName>Status</PropertyName>
      </TableColumnItem>
      <TableColumnItem>
        <PropertyName>Name</PropertyName>
      </TableColumnItem>
      <TableColumnItem>
        <PropertyName>DisplayName</PropertyName>
      </TableColumnItem>
    </TableColumnItems>
  </TableRowEntry>
</TableRowEntries>
</TableControl>
</View>

```

The following XML elements are used to define a list view:

- The [View](#) element is the parent element of the table view. (This is the same parent element for the list, wide, and custom control views.)
- The [Name](#) element specifies the name of the view. This element is required for all views.
- The [ViewSelectedBy](#) element defines the objects that use the view. This element is required.
- The [GroupBy](#) element (not shown in this example) defines when a new group of objects is displayed. A new group is started whenever the value of a specific property or script changes. This element is optional.
- The [Controls](#) element (not shown in this example) defines the custom controls that are defined by the table view. Controls give you a way to further specify how the data is displayed. This element is optional. A view can define its own custom controls, or it can use common controls that can be used by any view in the formatting file. For more information about custom controls, see [Creating Custom Controls](#).

- The [HideTableHeaders](#) element (not shown in this example) specifies that the table will not show any labels at the top of the table. This element is optional.
- The [TableControl](#) element that defines the header and row information of the table. Similar to all other views, a table view can display the values of object properties or values generated by scripts.

Defining Column Headers

1. The [TableHeaders](#) element and its child elements define what is displayed at the top of the table.
 2. The [TableColumnHeader](#) element defines what is displayed at the top of a column of the table. Specify these elements in the order that you want the headers displayed.
- There is no limit to the number of these elements that you can use, but the number of [TableColumnHeader](#) elements in your table view must equal the number of [TableRowEntry](#) elements that you use.
3. The [Label](#) element specifies the text that is displayed. This element is optional.
 4. The [Width](#) element specifies the width (in characters) of the column. This element is optional.
 5. The [Alignment](#) element specifies how the label is displayed. The label can be aligned to the left, to the right, or centered. This element is optional.

Defining the Table Rows

Table views can provide one or more definitions that specify what data is displayed in the rows of the table by using the child elements of the [TableRowEntries](#) element. Notice that you can specify multiple definitions for the rows of the table, but the headers for the rows remain the same, regardless of what row definition is used. Typically, a table will have only one definition.

In the following example, the view provides a single definition that displays the values of several properties of the [System.Diagnostics.Process](#) object. A table view can display the value of a property or the value of a script (not shown in the example) in its rows.

XML

```
<TableRowEntries>
  <TableRowEntry>
    <TableColumnItems>
      < TableColumnItem>
        <PropertyName>Status</PropertyName>
      </ TableColumnItem>
      < TableColumnItem>
        <PropertyName>Name</PropertyName>
      </ TableColumnItem>
      < TableColumnItem>
        <PropertyName>DisplayName</PropertyName>
      </ TableColumnItem>
    </TableColumnItems>
  </TableRowEntry>
</TableRowEntries>
```

The following XML elements can be used to provide definitions for a row:

- The **TableRowEntries** element and its child elements define what is displayed in the rows of the table.
- The **TableRowEntry** element provides a definition of the row. At least one **TableRowEntry** is required; however, there is no maximum limit to the number of elements that you can add. In most cases, a view will have only one definition.
- The **EntrySelectedBy** element specifies the objects that are displayed by a specific definition. This element is optional and is needed only when you define multiple **TableRowEntry** elements that display different objects.
- The **Wrap** element specifies that text that exceeds the column width is displayed on the next line. By default, text that exceeds the column width is truncated.
- The **TableColumnItems** element defines the properties or scripts whose values are displayed in the row.
- The **TableColumnItem** element defines the property or script whose value is displayed in the column of the row. A **TableColumnItem** element is required for each column of the row. The first entry is displayed in first column, the second entry in the second column, and so on.
- The **PropertyName** element specifies the property whose value is displayed in the row. You must specify either a property or a script, but you cannot specify both.
- The **ScriptBlock** element specifies the script whose value is displayed in the row. You must specify either a script or a property, but you cannot specify both.

- The [FormatString](#) element specifies a format pattern that defines how the property or script value is displayed. This element is optional.
- The [Alignment](#) element specifies how the value of the property or script is displayed. The value can be aligned to the left, to the right, or centered. This element is optional.

Defining the Objects That Use the Table View

There are two ways to define which .NET objects use the table view. You can use the [ViewSelectedBy](#) element to define the objects that can be displayed by all the definitions of the view, or you can use the [EntrySelectedBy](#) element to define which objects are displayed by a specific definition of the view. In most cases, a view has only one definition, so objects are typically defined by the [ViewSelectedBy](#) element.

The following example shows how to define the objects that are displayed by the table view using the [ViewSelectedBy](#) and [TypeName](#) elements. There is no limit to the number of [TypeName](#) elements that you can specify, and their order is not significant.

```
XML

<View>
  <Name>System.ServiceProcess.ServiceController</Name>
  <ViewSelectedBy>
    <TypeName>System.ServiceProcess.ServiceController</TypeName>
  </ViewSelectedBy>
  <TableControl>...</TableControl>
</View>
```

The following XML elements can be used to specify the objects that are used by the table view:

- The [ViewSelectedBy](#) element defines which objects are displayed by the list view.
- The [TypeName](#) element specifies the .NET object that is displayed by the view. The fully qualified .NET type name is required. You must specify at least one type or selection set for the view, but there is no maximum number of elements that can be specified.

The following example uses the [ViewSelectedBy](#) and [SelectionSetName](#) elements. Use selection sets where you have a related set of objects that are displayed using multiple views, such as when you define a list view and a table view for the same objects. For more information about how to create a selection set, see [Defining Selection Sets](#).

XML

```
<View>
  <Name>System.ServiceProcess.ServiceController</Name>
  <ViewSelectedBy>
    <SelectionSetName>.NET Type Set</SelectionSetName>
  </ViewSelectedBy>
  <TableControl>...</TableControl>
</View>
```

The following XML elements can be used to specify the objects that are used by the list view:

- The [ViewSelectedBy](#) element defines which objects are displayed by the list view.
- The [SelectionSetName](#) element specifies a set of objects that can be displayed by the view. You must specify at least one selection set or type for the view, but there is no maximum number of elements that can be specified.

The following example shows how to define the objects displayed by a specific definition of the table view using the [EntrySelectedBy](#) element. Using this element, you can specify the .NET type name of the object, a selection set of objects, or a selection condition that specifies when the definition is used. For more information about how to create a selection conditions, see [Defining Conditions for Displaying Data](#).

Note

When creating multiple definitions of the table view you cannot specify different column headers. You can specify only what is displayed in the rows of the table, such as what objects are displayed.

XML

```
<TableRowEntry>
  <EntrySelectedBy>
    <TypeName>.NET Type</TypeName>
  </EntrySelectedBy>
</TableRowEntry>
```

The following XML elements can be used to specify the objects that are used by a specific definition of the list view:

- The [EntrySelectedBy](#) element defines which objects are displayed by the definition.

- The [TypeName](#) element specifies the .NET object that is displayed by the definition. When using this element, the fully qualified .NET type name is required. You must specify at least one type, selection set, or selection condition for the definition, but there is no maximum number of elements that can be specified.
- The [SelectionSetName](#) element (not shown) specifies a set of objects that can be displayed by this definition. You must specify at least one type, selection set, or selection condition for the definition, but there is no maximum number of elements that can be specified.
- The [SelectionCondition](#) element (not shown) specifies a condition that must exist for this definition to be used. You must specify at least one type, selection set, or selection condition for the definition, but there is no maximum number of elements that can be specified. For more information about defining selection conditions, see [Defining Conditions for Displaying Data](#).

Using Format Strings

Formatting strings can be added to a view to further define how the data is displayed. The following example shows how to define a formatting string for the value of the [StartTime](#) property.

```
XML

< TableColumnItem >
  < PropertyName >StartTime</PropertyName>
  < FormatString >{0:MMM} {0:DD} {0:HH}:{0:MM}</FormatString>
</ TableColumnItem >
```

The following XML elements can be used to specify a format pattern:

- The [TableColumnItem](#) element defines the property or script whose value is displayed in the column of the row. A [TableColumnItem](#) element is required for each column of the row. The first entry is displayed in first column, the second entry in the second column, and so on.
- The [PropertyName](#) element specifies the property whose value is displayed in the row. You must specify either a property or a script, but you cannot specify both.
- The [FormatString](#) element specifies a format pattern that defines how the property or script value is displayed.

In the following example, the `ToString` method is called to format the value of the script. Scripts can call any method of an object. Therefore, if an object has a method, such as `ToString`, that has formatting parameters, the script can call that method to format the output value of the script.

XML

```
<ListItem>
  <ScriptBlock>
    [string]::Format("{0,-10} {1,-8}", $_.LastWriteTime.ToString("d"),
$_.LastWriteTime.ToString("t"))
  </ScriptBlock>
</ListItem>
```

The following XML element can be used to calling the `ToString` method:

- The `TableColumnItem` element defines the property or script whose value is displayed in the column of the row. A `TableColumnItem` element is required for each column of the row. The first entry is displayed in first column, the second entry in the second column, and so on.
- The `ScriptBlock` element specifies the script whose value is displayed in the row. You must specify either a script or a property, but you cannot specify both.

See Also

[Writing a PowerShell Formatting File](#)

Creating a List View

Article • 01/20/2022

A list view displays data in a single column (in sequential order). The data displayed in the list can be the value of a .NET property or the value of a script.

A List View Display

The following output shows how Windows PowerShell displays the properties of `System.ServiceProcess.ServiceController` objects that are returned by the [Get-Service](#) cmdlet. In this example, three objects were returned, with each object separated from the preceding object by a blank line.

PowerShell

```
Get-Service | Format-List
```

Output

```
Name          : AEADIFilters
DisplayName   : Andrea ADI Filters Service
Status        : Running
DependentServices : {}
ServicesDependedOn : {}
CanPauseAndContinue : False
CanShutdown    : False
CanStop       : True
ServiceType    : Win32OwnProcess

Name          : AeLookupSvc
DisplayName   : Application Experience
Status        : Running
DependentServices : {}
ServicesDependedOn : {}
CanPauseAndContinue : False
CanShutdown    : False
CanStop       : True
ServiceType    : Win32ShareProcess

Name          : AgereModemAudio
DisplayName   : Agere Modem Call Progress Audio
Status        : Running
DependentServices : {}
ServicesDependedOn : {}
CanPauseAndContinue : False
CanShutdown    : False
CanStop       : True
```

```
ServiceType      : Win32OwnProcess  
...
```

Defining the List View

The following XML shows the list view schema for displaying several properties of the `System.ServiceProcess.ServiceController` object. You must specify each property that you want displayed in the list view.

XML

```
<View>  
  <Name>System.ServiceProcess.ServiceController</Name>  
  <ViewSelectedBy>  
    <TypeName>System.ServiceProcess.ServiceController</TypeName>  
  </ViewSelectedBy>  
  <ListControl>  
    <ListEntries>  
      <ListEntry>  
        <ListItems>  
          <ListItem>  
            <PropertyName>Name</PropertyName>  
          </ListItem>  
          <ListItem>  
            <PropertyName>DisplayName</PropertyName>  
          </ListItem>  
          <ListItem>  
            <PropertyName>Status</PropertyName>  
          </ListItem>  
          <ListItem>  
            <PropertyName>ServiceType</PropertyName>  
          </ListItem>  
        </ListItems>  
      </ListEntry>  
    </ListEntries>  
  </ListControl>  
</View>
```

The following XML elements are used to define a list view:

- The `View` element is the parent element of the list view. (This is the same parent element for the table, wide, and custom control views.)
- The `Name` element specifies the name of the view. This element is required for all views.
- The `ViewSelectedBy` element defines the objects that use the view. This element is required.

- The [GroupBy](#) element defines when a new group of objects is displayed. A new group is started whenever the value of a specific property or script changes. This element is optional.
- The [Controls](#) element defines the custom controls that are defined by the list view. Controls give you a way to further specify how the data is displayed. This element is optional. A view can define its own custom controls, or it can use common controls that can be used by any view in the formatting file. For more information about custom controls, see [Creating Custom Controls](#).
- The [ListControl](#) element defines what is displayed in the view and how it is formatted. Similar to all other views, a list view can display the values of object properties or values generated by script.

For an example of a complete formatting file that defines a simple list view, see [List View \(Basic\)](#).

Providing Definitions for Your List View

List views can provide one or more definitions by using the child elements of the [ListControl](#) element. Typically, a view will have only one definition. In the following example, the view provides a single definition that displays several properties of the [System.Diagnostics.Process](#) object. A list view can display the value of a property or the value of a script (not shown in the example).

XML

```
<ListControl>
  <ListEntries>
    <ListEntry>
      <ListItems>
        <ListItem>
          <PropertyName>Name</PropertyName>
        </ListItem>
        <ListItem>
          <PropertyName>DisplayName</PropertyName>
        </ListItem>
        <ListItem>
          <PropertyName>Status</PropertyName>
        </ListItem>
        <ListItem>
          <PropertyName>ServiceType</PropertyName>
        </ListItem>
      </ListItems>
    </ListEntry>
  </ListEntries>
```

```
</ListControl>
```

The following XML elements can be used to provide definitions for a list view:

- The [ListControl](#) element and its child elements define what is displayed in the view.
- The [ListEntries](#) element provides the definitions of the view. In most cases, a view will have only one definition. This element is required.
- The [ListEntry](#) element provides a definition of the view. At least one [ListEntry](#) is required; however, there is no maximum limit to the number of elements that you can add. In most cases, a view will have only one definition.
- The [EntrySelectedBy](#) element specifies the objects that are displayed by a specific definition. This element is optional and is needed only when you define multiple [ListEntry](#) elements that display different objects.
- The [ListItems](#) element specifies the properties and scripts whose values are displayed in the rows of the list view.
- The [ListItem](#) element specifies a property or script whose value is displayed in a row of the list view. A list view must specify at least one property or script. There is no maximum limit to the number of rows that can be specified.
- The [PropertyName](#) element specifies the property whose value is displayed in the row. You must specify either a property or a script, but you cannot specify both.
- The [ScriptBlock](#) element specifies the script whose value is displayed in the row. You must specify either a script or a property, but you cannot specify both.
- The [Label](#) element specifies the label that is displayed to the left of the property or script value in the row. This element is optional. If a label is not specified, the name of the property or the script is displayed. For a complete example, see [List View \(Labels\)](#).
- The [ItemSelectionCondition](#) element specifies a condition that must exist for the row to be displayed. For more information about adding conditions to the list view, see [Defining Conditions for Displaying Data](#). This element is optional.
- The [FormatString](#) element specifies a pattern that is used to display the value of the property or script. This element is optional.

For an example of a complete formatting file that defines a simple list view, see [List View \(Basic\)](#).

Defining the Objects That Use the List View

There are two ways to define which .NET objects use the list view. You can use the [ViewSelectedBy](#) element to define the objects that can be displayed by all the definitions of the view, or you can use the [EntrySelectedBy](#) element to define which objects are displayed by a specific definition of the view. In most cases, a view has only one definition, so objects are typically defined by the [ViewSelectedBy](#) element.

The following example shows how to define the objects that are displayed by the list view using the [ViewSelectedBy](#) and [TypeName](#) elements. There is no limit to the number of [TypeName](#) elements that you can specify, and their order is not significant.

```
XML

<View>
  <Name>System.ServiceProcess.ServiceController</Name>
  <ViewSelectedBy>
    <TypeName>System.Diagnostics.Process</TypeName>
  </ViewSelectedBy>
  <ListControl>...</ListControl>
</View>
```

The following XML elements can be used to specify the objects that are used by the list view:

- The [ViewSelectedBy](#) element defines which objects are displayed by the list view.
- The [TypeName](#) element specifies the .NET object that is displayed by the view. The fully qualified .NET type name is required. You must specify at least one type or selection set for the view, but there is no maximum number of elements that can be specified.

For an example of a complete formatting file, see [List View \(Basic\)](#).

The following example uses the [ViewSelectedBy](#) and [SelectionSetName](#) elements. Use selection sets where you have a related set of objects that are displayed using multiple views, such as when you define a list view and a table view for the same objects. For more information about how to create a selection set, see [Defining Selection Sets](#).

```
XML

<View>
  <Name>System.ServiceProcess.ServiceController</Name>
  <ViewSelectedBy>
    <SelectionSetName>.NET Type Set</SelectionSetName>
  </ViewSelectedBy>
```

```
<ListControl>...</ListControl>
</View>
```

The following XML elements can be used to specify the objects that are used by the list view:

- The [ViewSelectedBy](#) element defines which objects are displayed by the list view.
- The [SelectionSetName](#) element specifies a set of objects that can be displayed by the view. You must specify at least one selection set or type for the view, but there is no maximum number of elements that can be specified.

The following example shows how to define the objects displayed by a specific definition of the list view using the [EntrySelectedBy](#) element. Using this element, you can specify the .NET type name of the object, a selection set of objects, or a selection condition that specifies when the definition is used. For more information about how to create a selection conditions, see [Defining Conditions for Displaying Data](#).

XML

```
<ListEntry>
  <EntrySelectedBy>
    <TypeName>.NET Type</TypeName>
  </EntrySelectedBy>
</ListEntry>
```

The following XML elements can be used to specify the objects that are used by a specific definition of the list view:

- The [EntrySelectedBy](#) element defines which objects are displayed by the definition.
- The [TypeName](#) element specifies the .NET object that is displayed by the definition. When using this element, the fully qualified .NET type name is required. You must specify at least one type, selection set, or selection condition for the definition, but there is no maximum number of elements that can be specified.
- The [SelectionSetName](#) element (not shown) specifies a set of objects that can be displayed by this definition. You must specify at least one type, selection set, or selection condition for the definition, but there is no maximum number of elements that can be specified.
- The [SelectionCondition](#) element (not shown) specifies a condition that must exist for this definition to be used. You must specify at least one type, selection set, or selection condition for the definition, but there is no maximum number of

elements that can be specified. For more information about defining selection conditions, see [Defining Conditions for Displaying Data](#).

Displaying Groups of Objects in a List View

You can separate the objects that are displayed by the list view into groups. This does not mean that you define a group, only that Windows PowerShell starts a new group whenever the value of a specific property or script changes. In the following example, a new group is started whenever the value of the [System.ServiceProcess.ServiceController.ServiceType](#) property changes.

XML

```
<GroupBy>
  <Label>Service Type</Label>
  <PropertyName>ServiceType</PropertyName>
</GroupBy>
```

The following XML elements are used to define when a group is started:

- The [GroupBy](#) element defines the property or script that starts the new group and defines how the group is displayed.
- The [PropertyName](#) element specifies the property that starts a new group whenever its value changes. You must specify a property or script to start the group, but you cannot specify both.
- The [ScriptBlock](#) element specifies the script that starts a new group whenever its value changes. You must specify a script or property to start the group, but you cannot specify both.
- The [Label](#) element defines a label that is displayed at the beginning of each group. In addition to the text specified by this element, Windows PowerShell displays the value that triggered the new group and adds a blank line before and after the label. This element is optional.
- The [CustomControl](#) element defines a control that is used to display the data. This element is optional.
- The [CustomControlName](#) element specifies a common or view control that is used to display the data. This element is optional.

For an example of a complete formatting file that defines groups, see [List View \(GroupBy\)](#).

Using Format Strings

Formatting strings can be added to a view to further define how the data is displayed. The following example shows how to define a formatting string for the value of the `StartTime` property.

XML

```
<ListItem>
  <PropertyName>StartTime</PropertyName>
  <FormatString>{0:MMM} {0:DD} {0:HH}:{0:MM}</FormatString>
</ListItem>
```

The following XML elements can be used to specify a format pattern:

- The `ListItem` element specifies the data that is displayed by the view.
- The `PropertyName` element specifies the property whose value is displayed by the view. You must specify either a property or a script, but you cannot specify both.
- The `FormatString` element specifies a format pattern that defines how the property or script value is displayed in the view.
- The `ScriptBlock` element (not shown) specifies the script whose value is displayed by the view. You must specify either a script or a property, but you cannot specify both.

In the following example, the `ToString` method is called to format the value of the script. Scripts can call any method of an object. Therefore, if an object has a method, such as `ToString`, that has formatting parameters, the script can call that method to format the output value of the script.

XML

```
<ListItem>
  <ScriptBlock>
    [string]::Format("{0,-10} {1,-8}", $_.LastWriteTime.ToString("d"),
$_.LastWriteTime.ToString("t"))
  </ScriptBlock>
</ListItem>
```

The following XML element can be used to calling the `ToString` method:

- The [ListItem](#) element specifies the data that is displayed by the view.
- The [ScriptBlock](#) element (not shown) specifies the script whose value is displayed by the view. You must specify either a script or a property, but you cannot specify both.

See Also

[Writing a Windows PowerShell Cmdlet](#)

Creating a Wide View

Article • 03/13/2023

A wide view displays a single value for each object that's displayed. The displayed value can be the value of a .NET object property or the value of a script. By default, there is no label or header for this view.

A Wide View Display

The following example shows how Windows PowerShell displays the `System.Diagnostics.Process` object that's returned by the `Get-Process` cmdlet when its output is piped to the `Format-Wide` cmdlet. (By default, the `Get-Process` cmdlet returns a table view.) In this example, the two columns are used to display the name of the process for each returned object. The name of the object's property isn't displayed, only the value of the property.

```
PowerShell
Get-Process | Format-Wide

Output
AEADISRV      agrsmsvc
Ati2evxx     Ati2evxx
audiogd       CCC
CcmExec      communicator
Crypserv     csrss
csrss        DevDtct2
DM1Service   dpupdchk
dwm          DxStudio
EXCEL        explorer
GoogleToolbarNotifier GrooveMonitor
hpqwmiepx   hpservice
Idle         InoRpc
InoRT        InoTask
ipoint       lsass
lsm          MOM
MSASCui     notepad
...
...
```

Defining the Wide View

The following XML shows the wide view schema for the [System.Diagnostics.Process](#) object.

```
XML

<View>
  <Name>process</Name>
  <ViewSelectedBy>
    <TypeName>System.Diagnostics.Process</TypeName>
  </ViewSelectedBy>
  <GroupBy>...</GroupBy>
  <Controls>...</Controls>
  <WideControl>
    <WideEntries>
      <WideEntry>
        <WideItem>
          <PropertyName>ProcessName</PropertyName>
        </WideItem>
      </WideEntry>
    </WideEntries>
  </WideControl>
</View>
```

The following XML elements are used to define a wide view:

- The [View](#) element is the parent element of the wide view. (This is the same parent element for the table, list, and custom control views.)
- The [Name](#) element specifies the name of the view. This element is required for all views.
- The [ViewSelectedBy](#) element defines the objects that use the view. This element is required.
- The [GroupBy](#) element defines when a new group of objects is displayed. A new group is started whenever the value of a specific property or script changes. This element is optional.
- The [Controls](#) elements defines the custom controls that are defined by the wide view. Controls give you a way to further specify how the data is displayed. This element is optional. A view can define its own custom controls, or it can use common controls that can be used by any view in the formatting file. For more information about custom controls, see [Creating Custom Controls](#).
- The [WideControl](#) element and its child elements define what's displayed in the view. In the preceding example, the view is designed to display the [System.Diagnostics.Process.ProcessName](#) property.

For an example of a complete formatting file that defines a simple wide view, see [Wide View \(Basic\)](#).

Providing Definitions for Your Wide View

Wide views can provide one or more definitions by using the child elements of the [WideControl](#) element. Typically, a view will have only one definition. In the following example, the view provides a single definition that displays the [System.Diagnostics.Process.ProcessName](#) property. A wide view can display the value of a property or the value of a script (not shown in the example).

XML

```
<WideControl>
  <AutoSize/>
  <ColumnNumber></ColumnNumber>
  <WideEntries>
    <WideEntry>
      <WideItem>
        <PropertyName>ProcessName</PropertyName>
      </WideItem>
    </WideEntry>
  </WideEntries>
</WideControl>
```

The following XML elements can be used to provide definitions for a wide view:

- The [WideControl](#) element and its child elements define what's displayed in the view.
- The [AutoSize](#) element specifies whether the column size and the number of columns are adjusted based on the size of the data. This element is optional.
- The [ColumnName](#) element specifies the number of columns displayed in the wide view. This element is optional.
- The [WideEntries](#) element provides the definitions of the view. In most cases, a view will have only one definition. This element is required.
- The [WideEntry](#) element provides a definition of the view. At least one [WideEntry](#) is required; however, there is no maximum limit to the number of elements that you can add. In most cases, a view will have only one definition.
- The [EntrySelectedBy](#) element specifies the objects that are displayed by a specific definition. This element is optional and is needed only when you define multiple [WideEntry](#) elements that display different objects.
- The [WideItem](#) element specifies the data that's displayed by the view. In contrast to other types of views, a wide control can display only one item.
- The [PropertyName](#) element specifies the property whose value is displayed by the view. You must specify either a property or a script, but you can't specify both.
- The [ScriptBlock](#) element specifies the script whose value is displayed by the view. You must specify either a script or a property, but you can't specify both.

- The [FormatString](#) element specifies a pattern that's used to display the data. This element is optional.

For an example of a complete formatting file that defines a wide view definition, see [Wide View \(Basic\)](#).

Defining the Objects That Use the Wide View

There are two ways to define which .NET objects use the wide view. You can use the [ViewSelectedBy](#) element to define the objects that can be displayed by all the definitions of the view, or you can use the [EntrySelectedBy](#) element to define which objects are displayed by a specific definition of the view. In most cases, a view has only one definition, so objects are typically defined by the [ViewSelectedBy](#) element.

The following example shows how to define the objects that are displayed by the wide view using the [ViewSelectedBy](#) and [TypeName](#) elements. There is no limit to the number of [TypeName](#) elements that you can specify, and their order isn't significant.

XML

```
<View>
  <Name>System.ServiceProcess.ServiceController</Name>
  <ViewSelectedBy>
    <TypeName>System.Diagnostics.Process</TypeName>
  </ViewSelectedBy>
  <WideControl>...</WideControl>
</View>
```

The following XML elements can be used to specify the objects that are used by the wide view:

- The [ViewSelectedBy](#) element defines which objects are displayed by the wide view.
- The [TypeName](#) element specifies the .NET that's displayed by the view. The fully qualified .NET type name is required. You must specify at least one type or selection set for the view, but there is no maximum number of elements that can be specified.

For an example of a complete formatting file, see [Wide View \(Basic\)](#).

The following example uses the [ViewSelectedBy](#) and [SelectionSetName](#) elements. Use selection sets where you have a related set of objects that are displayed using multiple views, such as when you define a wide view and a table view for the same objects. For more information about how to create a selection set, see [Defining Selection Sets](#).

XML

```
<View>
  <Name>System.ServiceProcess.ServiceController</Name>
  <ViewSelectedBy>
    <SelectionSetName>.NET Type Set</SelectionSetName>
  </ViewSelectedBy>
  <WideControl>...</WideControl>
</View>
```

The following XML elements can be used to specify the objects that are used by the wide view:

- The [ViewSelectedBy](#) element defines which objects are displayed by the wide view.
- The [SelectionSetName](#) element specifies a set of objects that can be displayed by the view. You must specify at least one selection set or type for the view, but there is no maximum number of elements that can be specified.

The following example shows how to define the objects displayed by a specific definition of the wide view using the [EntrySelectedBy](#) element. Using this element, you can specify the .NET type name of the object, a selection set of objects, or a selection condition that specifies when the definition is used. For more information about how to create a selection conditions, see [Defining Conditions for Displaying Data](#).

XML

```
<WideEntry>
  <EntrySelectedBy>
    <TypeName>.NET Type</TypeName>
  </EntrySelectedBy>
</WideEntry>
```

The following XML elements can be used to specify the objects that are used by a specific definition of the wide view:

- The [EntrySelectedBy](#) element defines which objects are displayed by the definition.
- The [TypeName](#) element specifies the .NET that's displayed by the definition. When using this element the fully qualified .NET type name is required. You must specify at least one type, selection set, or selection condition for the definition, but there is no maximum number of elements that can be specified.
- The [SelectionSetName](#) element (not shown) specifies a set of objects that can be displayed by this definition. You must specify at least one type, selection set, or selection condition for the definition, but there is no maximum number of elements that can be specified.

- The [SelectionCondition](#) element (not shown) specifies a condition that must exist for this definition to be used. You must specify at least one type, selection set, or selection condition for the definition, but there is no maximum number of elements that can be specified. For more information about defining selection conditions, see [Defining Conditions for Displaying Data](#).

Displaying Groups of objects in a Wide View

You can separate the objects that are displayed by the wide view into groups. This doesn't mean that you define a group, only that Windows PowerShell starts a new group whenever the value of a specific property or script changes. In the following example, a new group is started whenever the value of the [System.ServiceProcess.ServiceController.ServiceType](#) property changes.

XML

```
<GroupBy>
  <Label>Service Type</Label>
  <PropertyName>ServiceType</PropertyName>
</GroupBy>
```

The following XML elements are used to define when a group is started:

- The [GroupBy](#) element defines the property or script that starts the new group and defines how the group is displayed.
- The [PropertyName](#) element specifies the property that starts a new group whenever its value changes. You must specify a property or script to start the group, but you can't specify both.
- The [ScriptBlock](#) element specifies the script that starts a new group whenever its value changes. You must specify a script or property to start the group, but you can't specify both.
- The [Label](#) element defines a label that's displayed at the beginning of each group. In addition to the text specified by this element, Windows PowerShell displays the value that triggered the new group and adds a blank line before and after the label. This element is optional.
- The [CustomControl](#) element defines a control that's used to display the data. This element is optional.
- The [CustomControlName](#) element specifies a common or view control that's used to display the data. This element is optional.

For an example of a complete formatting file that defines groups, see [Wide View \(GroupBy\)](#).

Using Format Strings

Formatting strings can be added to a wide view to further define how the data is displayed. The following example shows how to define a formatting string for the value of the `StartTime` property.

XML

```
<WideItem>
  <PropertyName>StartTime</PropertyName>
  <FormatString>{0:MMM} {0:DD} {0:HH}:{0:MM}</FormatString>
</WideItem>
```

The following XML elements can be used to specify a format pattern:

- The `WideItem` element specifies the data that's displayed by the view.
- The `PropertyName` element specifies the property whose value is displayed by the view. You must specify either a property or a script, but you can't specify both.
- The `FormatString` element specifies a format pattern that defines how the property or script value is displayed in the view
- The `ScriptBlock` element (not shown) specifies the script whose value is displayed by the view. You must specify either a script or a property, but you can't specify both.

In the following example, the `ToString` method is called to format the value of the script. Scripts can call any method of an object. Therefore, if an object has a method, such as `ToString`, that has formatting parameters, the script can call that method to format the output value of the script.

XML

```
<WideItem>
  <ScriptBlock>
    [string]::Format("{0,-10} {1,-8}", $_.LastWriteTime.ToString("d"),
    $_.LastWriteTime.ToString("t"))
  </ScriptBlock>
</WideItem>
```

The following XML element can be used to calling the `ToString` method:

- The `WideItem` element specifies the data that's displayed by the view.

- The [ScriptBlock](#) element (not shown) specifies the script whose value is displayed by the view. You must specify either a script or a property, but you can't specify both.

See Also

- [Wide View \(Basic\)](#)
- [Wide View \(GroupBy\)](#)
- [Writing a PowerShell Formatting File](#)

Creating Custom Controls

Article • 09/17/2021

Custom controls are the most flexible components of a formatting file. Unlike table, list, and wide views that define a formal structure of data, such as a table of data, custom controls allow you to define how an individual piece of data is displayed. You can define a common set of custom controls that are available to all the views of the formatting file, you can define custom controls that are available to a specific view, or you can define a set of controls that are available to a group of objects.

Custom Control Example

The following example shows a custom control that is defined in the Certificates.Format.ps1xml file. This custom control is used to separate the [System.Management.Automation.Signature](#) objects displayed in a table view.

XML

```
<Controls>
  <Control>
    <Name>SignatureTypes-GroupingFormat</Name>
    <CustomControl>
      <CustomEntries>
        <CustomEntry>
          <CustomItem>
            <Frame>
              <LeftIndent>4</LeftIndent>
              <CustomItem>
                <Text AssemblyName="System.Management.Automation"
BaseName="FileSystemProviderStrings"
                  ResourceId="DirectoryDisplayGrouping"/>
                <ExpressionBinding>
                  <ScriptBlock>Split-Path $_.Path</ScriptBlock>
                </ExpressionBinding>
                <NewLine/>
              </CustomItem>
            </Frame>
          </CustomItem>
        </CustomEntry>
      </CustomEntries>
    </CustomControl>
  </Control>
</Controls>
```

See Also

[Writing a PowerShell Formatting File](#)

Loading and Exporting Formatting Data

Article • 03/07/2023

Once you've created your formatting file, you need to update the format data of the session by loading your files into the current session. PowerShell loads a predefined set of formats. Once the format data of the current session is updated, PowerShell uses that data to display the .NET objects associated with the views defined in the loaded formats. There's no limit to the number of formats that you can load into the current session. You can also export the format data in the current session back to a formatting file.

Loading format data

Formatting files can be loaded into the current session using the following methods:

- You can import the formatting file into the current session from the command line. Use the [Update-FormatData](#) cmdlet as described in the following procedure.
- You can create a module manifest that references your formatting file. Modules allow you to package your formatting files for distribution. Use the [New-ModuleManifest](#) cmdlet to create the manifest, and the [Import-Module](#) cmdlet to load the module into the current session. For more information about modules, see [Writing a Windows PowerShell Module](#).
- You can create a snap-in that references your formatting file. Use the [System.Management.Automation.PSSnapIn.Formats](#) to reference your formatting files. However, best practice recommendation is to use modules to package cmdlets and associated formatting and types files.
- If you're invoking commands programmatically, you can add formatting files to the initial session state of the runspace where the commands are run. For more information, see the [System.Management.Automation.Runspaces.SessionStateFormatEntry](#) class.

When a formatting file is loaded, it's added to an internal list that PowerShell uses to choose the view used when displaying objects in the host. You can prepend your formatting file to the beginning of the list, or you can append it to the end of the list.

Knowing where your formatting file is added to this list is important.

- If you're loading a formatting file that defines the only view for an object, you can use any of the methods described previously.

- If you're loading a formatting file that defines a view for an object that has an existing view defined, it must be added to the beginning of the list. You must use the [Update-FormatData](#) cmdlet and prepend your file to the beginning of the list.

Storing Your Formatting File

You can store formatting files anywhere on disk. However, it's recommended that you store them in the same folder as your profile script.

Use the following command to determine the location of your profile script.

```
PowerShell
```

```
Split-Path -Path $PROFILE -Parent
```

Loading a format file

1. Store your formatting file to disk.
2. Run the [Update-FormatData](#) cmdlet using one of the following commands.

If you're changing how an object is displayed, use the following command to add your formatting file to the front of the list.

```
PowerShell
```

```
Update-FormatData -PrependPath PathToFormattingFile
```

Use the following command to add your formatting file to the end of the list.

```
PowerShell
```

```
Update-FormatData -AppendPath PathToFormattingFile
```

(!) Note

Once format data has been loaded in a session it can't be removed. You must open a new session without the format data loaded.

Exporting format data

PowerShell includes format definitions for many .NET types. You can use the [Get-FormatData](#) cmdlet to view the format data that's loaded in the current session. You can export the format data for a type to a file using the [Export-FormatData](#) cmdlet.

The following commands export the format data for the `System.Guid` type to a file named `System.Guid.format.ps1xml` in the current directory.

PowerShell

```
Get-FormatData System.Guid | Export-FormatData -Path  
./System.Guid.format.ps1xml  
Get-Content ./System.Guid.format.ps1xml
```

Output

```
<?xml version="1.0" encoding="utf-8"?>  
<Configuration>  
  <ViewDefinitions>  
    <View>  
      <Name>System.Guid</Name>  
      <ViewSelectedBy>  
        <TypeName>System.Guid</TypeName>  
      </ViewSelectedBy>  
      <TableControl>  
        <TableHeaders />  
        <TableRowEntries>  
          <TableRowEntry>  
            <TableColumnItems>  
              <TableColumnItem>  
                <PropertyName>Guid</PropertyName>  
              </TableColumnItem>  
            </TableColumnItems>  
          </TableRowEntry>  
        </TableRowEntries>  
      </TableControl>  
    </View>  
  </ViewDefinitions>  
</Configuration>
```

You can edit the exported file to create a custom format definition for that type.

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Defining Selection Sets

Article • 09/17/2021

When creating multiple views and controls, you can define sets of objects that are referred to as selection sets. A selection set enables you to define the objects one time, without having to define them repeatedly for each view or control. Typically, selection sets are used when you have a set of related .NET objects. For example, The `FileSystem` formatting file (`FileSystem.format.ps1xml`) defines a selection set of the file system types that several views use.

Where Selection Sets are Defined and Referenced

You define selection sets as part of the common data that can be used by all the views and controls defined in the formatting file. The following example shows how to define three selection sets.

XML

```
<Configuration>
  <SelectionSets>
    <SelectionSet>...</SelectionSet>
    <SelectionSet>...</SelectionSet>
    <SelectionSet>...</SelectionSet>
  </SelectionSets>
</Configuration>
```

You can reference a selection sets in the following ways:

- Each view has a `ViewSelectedBy` element that defines which objects are displayed by using the view. The `ViewSelectedBy` element has a `SelectionSetName` child element that specifies the selection set that all the definitions of the view use. There is no restriction on the number of selection sets that you can reference from a view.
- In each definition of a view or control, the `EntrySelectedBy` element defines which objects are displayed by using that definition. Typically a view or control has only one definition so the objects are defined by the `ViewSelectedBy` element. The `EntrySelectedBy` element of the definition has a `SelectionSetName` child element that specifies the selection set. If you specify the selection set for a definition, you cannot specify any of the other child elements of the `EntrySelectedBy` element.

- In each definition of a view or control, the `SelectionCondition` element can be used to specify a condition for when the definition is used. The `SelectionCondition` element has a `SelectionSetName` child element that specifies the selection set that triggers the condition. The condition is triggered when any of the objects defined in the selection set are displayed. For more information about how to set these conditions, see [Defining Conditions for when Data is Displayed](#).

Selection Set Example

The following example shows a selection set that is taken directly from the `FileSystem` formatting file provided by Windows PowerShell. For more information about other Windows PowerShell formatting files, see [Windows PowerShell Formatting Files](#).

XML

```
<SelectionSets>
  <SelectionSet>
    <Name>FileSystemTypes</Name>
    <Types>
      <TypeName>System.IO.DirectoryInfo</TypeName>
      <TypeName>System.IO.FileInfo</TypeName>
      <TypeName>Deserialized.System.IO.DirectoryInfo</TypeName>
      <TypeName>Deserialized.System.IO.FileInfo</TypeName>
    </Types>
  </SelectionSet>
</SelectionSets>
```

The previous selection set is referenced in the `ViewSelectedBy` element of a table view.

XML

```
<ViewDefinitions>
  <View>
    <Name>Files</Name>
    <ViewSelectedBy>
      <SelectionSetName>FileSystemTypes</SelectionSetName>
    </ViewSelectedBy>
    <TableControl>...</TableControl>
  </View>
</ViewDefinitions>
```

XML Elements

There is no limit to the number of selection sets that you can define. The following XML elements are used to create a selection set.

- The [SelectionSets](#) element defines the sets of .NET objects that are referenced by the views and controls of the formatting file.
- The [SelectionSet](#) element defines a single set of .NET objects.
- The [Name](#) element specifies the name that is used to reference the selection set.
- The [Types](#) element specifies the .NET types of the objects of the selection set.
(Within formatting files, objects are specified by their .NET type.)

The following XML elements are used to specify a selection set.

- The following element specifies the selection set to use in all the definitions of the view:
 - [SelectionSetName Element for ViewSelectedBy \(Format\)](#)
 - [SelectionSetName Element for EntrySelectedBy for GroupBy \(Format\)](#)
- The following elements specify the selection set used by a single view definition:
 - [SelectionSetName Element for EntrySelectedBy for ListControl \(Format\)](#)
 - [SelectionSetName Element for EntrySelectedBy for TableControl \(Format\)](#)
 - [SelectionSetName Element for EntrySelectedBy for WideControl \(Format\)](#)
 - [SelectionSetName Element for EntrySelectedBy for CustomControl for View \(Format\)](#)
- The following elements specify the selection set used by common and view control definitions:
 - [SelectionSetName Element for EntrySelectedBy for Controls for View \(Format\)](#)
 - [SelectionSetName Element for EntrySelectedBy for Controls for Configuration \(Format\)](#)
- The following elements specify the selection set used when you define which object to expand:
 - [SelectionSetName Element for EntrySelectedBy for EnumerableExpansion \(Format\)](#)
- The following elements specify the selection set used by selection conditions.

- [SelectionSetName Element for SelectionCondition for Controls for Configuration \(Format\)](#)
- [SelectionSetName Element for SelectionCondition for Controls for View \(Format\)](#)
- [SelectionSetName Element for SelectionCondition for CustomControl for View \(Format\)](#)
- [SelectionSetName Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion \(Format\)](#)
- [SelectionSetName Element for SelectionCondition for EntrySelectedBy for ListEntry \(Format\)](#)
- [SelectionSetName Element for SelectionCondition for EntrySelectedBy for TableControl \(Format\)](#)
- [SelectionSetName Element for SelectionCondition for EntrySelectedBy for WideEntry \(Format\)](#)
- [SelectionSetName Element for SelectionCondition for GroupBy \(Format\)](#)

See Also

[SelectionSets](#)

[SelectionSet](#)

[Name](#)

[Types](#)

[PowerShell Formatting Files](#)

[Defining Conditions for when Data is Displayed](#)

[Writing a PowerShell Formatting and Types File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

Defining Conditions for Displaying Data

Article • 09/17/2021

When defining what data is displayed by a view or a control, you can specify a condition that must exist for the data to be displayed. The condition can be triggered by a specific property, or when a script or property value evaluates to `true`. When the selection condition is met, the definition of the view or control is used.

Specifying a Selection Condition for a Definition

When creating a definition for a view or control, the `EntrySelectedBy` element is used to specify which objects will use the definition or what condition must exist for the definition to be used. The condition is specified by the `SelectionCondition` element.

In the following example, a selection condition is specified for a definition of a table view. In this example, the definition is used only when the specified script is evaluated to `true`.

XML

```
<TableRowEntry>
  <EntrySelectedBy>
    <SelectionCondition>
      <ScriptBlock>ScriptToEvaluate</ScriptBlock>
    </SelectionCondition>
  </EntrySelectedBy>
  <TableColumnItems>
  </TableColumnItems>
</TableRowEntry>
```

There is no limit to the number of selection conditions that you can specify for a definition of a view or control. The only requirements are the following:

- The selection condition must specify one property name or script to trigger the condition, but cannot specify both.
- The selection condition can specify any number of .NET types or selection sets, but cannot specify both.

Specifying a Selection Condition for an Item

You can also specify when an item of a list view or control is used by including the `ItemSelectionCondition` element in the item definition. In the following example, a selection condition is specified for an item of a list view. In this example, the item is used only when the script is evaluated to `true`.

XML

```
<ListItem>
  <ItemSelectionCondition>
    <ScriptBlock>ScriptToEvaluate</ScriptBlock>
  </ItemSelectionCondition>
</ListItem>
```

You can specify only one selection condition for an item. And the condition must specify one property name or script to trigger the condition, but cannot specify both.

XML Elements

The following XML elements are used to create a selection condition.

- The following elements specify selection conditions for view definitions:
 - [SelectionCondition Element for EntrySelectedBy for TableControl \(Format\)](#)
 - [SelectionCondition Element for EntrySelectedBy for ListControl \(Format\)](#)
 - [SelectionCondition Element for EntrySelectedBy for WideControl \(Format\)](#)
 - [SelectionCondition Element for EntrySelectedBy for CustomControl \(Format\)](#)
- The following elements specify selection conditions for common and view control definitions:
 - [SelectionCondition Element for EntrySelectedBy for Controls for Configuration \(Format\)](#)
 - [SelectionCondition Element for EntrySelectedBy for Controls for View \(Format\)](#)
- The following element specifies the selection condition for expanding collection objects:
 - [SelectionCondition Element for EntrySelectedBy for EnumerableExpansion \(Format\)](#)

- The following element specifies the selection condition for displaying a new group of data:
 - [SelectionCondition Element for EntrySelectedBy for GroupBy \(Format\)](#)
- The following element specifies an item selection condition for a list view:
 - [ItemSelectionCondition Element for ListItem for ListControl \(Format\)](#)
- The following elements specify an item selection condition for controls:
 - [ItemSelectionCondition Element for ExpressionBinding for Controls for Configuration \(Format\)](#)
 - [ItemSelectionCondition Element for ExpressionBinding for Controls for View \(Format\)](#)
 - [ItemSelectionCondition Element for ExpressionBinding for CustomControl \(Format\)](#)

See Also

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Formatting Displayed Data

Article • 03/13/2023

You can specify how the individual data points in your List, Table, or Wide view are displayed. You can use the `FormatString` element when defining the items of your view, or you can use the `ScriptBlock` element to call the `FormatString` method on the data.

Using the `FormatString` Element

In the following example the value of the `TotalProcessorTime` property of the `System.Diagnostics.Process` object is formatted using the `FormatString` element. the `TotalProcessorTime` property

XML

```
<TableColumnItem>
  <PropertyName>TotalProcessorTime</PropertyName>
  <FormatString>{0:MMM}{0:dd}{0:HH}:{0:mm}</FormatString>
</TableColumnItem>
```



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Windows PowerShell Formatting Files

Article • 09/17/2021

Windows PowerShell provides several formatting files (.format.ps1xml) that are located in the installation directory (`$PSHOME`). Each of these files defines the default display for a specific set of .NET objects. These files should never be changed. However, you can use them as a reference for creating your own custom formatting files.

`Certificate.Format.ps1xml` Defines the display of objects in the Certificate store such as x.509 certificates and certificate stores.

`DotNetTypes.Format.ps1xml` Defines the display of miscellaneous .NET objects such as `CultureInfo`, `FileVersionInfo`, and `EventLogEntry` objects.

`FileSystem.Format.ps1xml` Defines the display of file system objects such as file and directory objects.

`Help.Format.ps1xml` Defines the different views used by the [Get-Help](#) cmdlet, such as the detailed, full, parameters, and example views.

`PowerShellCore.Format.ps1xml` Defines the display of the objects generated by Windows PowerShell core cmdlets, such as the objects returned by the [Get-Member](#) and [Get-History](#) cmdlets.

`PowerShellTrace.Format.ps1xml` Defines the display of trace objects such as those generated by the [Trace-Command](#) cmdlet.

`Registry.Format.ps1xml` Defines the display of registry objects such as key and entry objects.

See Also

[Writing a Windows PowerShell Cmdlet](#)

How to Create a Formatting File (.format.ps1xml)

Article • 09/17/2021

This topic describes how to create a formatting file (.format.ps1xml).

ⓘ Note

You can also create a formatting file by making a copy of one of the files provided by Windows PowerShell. If you make a copy of an existing file, delete the existing digital signature, and add your own signature to the new file.

Create a .format.ps1xml file.

1. Create a text file (.txt) using a text editor such as Notepad.

2. Copy the following lines into the formatting file.

XML

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
<ViewDefinitions>
</ViewDefinitions>
</Configuration>
```

- The `<Configuration></Configuration>` tags define the root `Configuration` node. All additional XML tags will be enclosed within this node.
- The `<ViewDefinitions></ViewDefinitions>` tags define the `ViewDefinitions` node. All views are defined within this node.

3. Save the file to the Windows PowerShell installation folder, to your module folder, or to a subfolder of the module folder. Use the following name format when you save the file: `MyFile.format.ps1xml`. Formatting files must use the `.format.ps1xml` extension.

You are now ready to add views to the formatting file. There is no limit to the number of views that can be defined in a formatting file. You can add a single view for each object, multiple views for the same object, or a single view that is used by multiple objects.

See Also

[Writing a Windows PowerShell Formatting and Types File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Wide View (Basic)

Article • 03/24/2025

This example shows how to implement a basic wide view that displays the [System.ServiceProcess.ServiceController](#) objects returned by the `Get-Service` cmdlet. For more information about the components of a wide view, see [Creating a Wide View](#).

Load this formatting file

1. Copy the XML from the Example section of this topic into a text file.
2. Save the text file. Be sure to add the `format.ps1xml` extension to the file to identify it as a formatting file.
3. Open Windows PowerShell, and run the following command to load the formatting file into the current session: `Update-FormatData -PrependPath <PathToFormattingFile>`.

⚠ Warning

This formatting file defines the display of an object that is already defined by a Windows PowerShell formatting file. You must use the `PrependPath` parameter when you run the cmdlet, and you cannot load this formatting file as a module.

Demonstrates

This formatting file demonstrates the following XML elements:

- The [Name](#) element for the view.
- The [ViewSelectedBy](#) element that defines what objects are displayed by the view.
- The [WideItem](#) element that defines what property is displayed by the view.

Example

The following XML defines a wide view that displays the value of the [System.ServiceProcess.ServiceController.ServiceName](#) property.

XML

```
<?xml version="1.0" encoding="utf-8" ?>

<Configuration>
  <ViewDefinitions>
    <View>
      <Name>ServiceWideView</Name>
      <ViewSelectedBy>
        <TypeName>System.ServiceProcess.ServiceController</TypeName>
      </ViewSelectedBy>
      <WideControl>
        <WideEntries>
          <WideEntry>
            <WideItem>
              <PropertyName>ServiceName</PropertyName>
            </WideItem>
          </WideEntry>
        </WideEntries>
      </WideControl>
    </View>
  </ViewDefinitions>
</Configuration>
```

The following example shows how Windows PowerShell displays the `System.ServiceProcess.ServiceController` objects after this format file is loaded.

PowerShell

```
Get-Service f*
```

Output

Fax	FCSAM
fdPHost	FDResPub
FontCache	FontCache3.0.0.0
FSysAgent	FwcAgent

See Also

[Examples of Formatting Files](#)

[Writing a PowerShell Formatting File](#)

Wide View (GroupBy)

Article • 03/24/2025

This example shows how to implement a wide view that displays groups of [System.ServiceProcess.ServiceController](#) objects returned by the `Get-Service` cmdlet. For more information about the components of a wide view, see [Creating a Wide View](#).

Load this formatting file

1. Copy the XML from the Example section of this topic into a text file.
2. Save the text file. Be sure to add the `.ps1xml` extension to the file to identify it as a formatting file.
3. Open Windows PowerShell, and run the following command to load the formatting file into the current session: `Update-FormatData -PrependPath <Path to file>`.

⚠ Warning

This formatting file defines the display of an object that is already defined by a Windows PowerShell formatting files. You must use the `PrependPath` parameter when you run the cmdlet, and you cannot load this formatting file as a module.

Demonstrates

This formatting file demonstrates the following XML elements:

- The [Name](#) element for the view.
- The [ViewSelectedBy](#) element that defines what objects are displayed by the view.
- The [GroupBy](#) element that defines when a new group is displayed.
- The [WideItem](#) element that defines what property is displayed by the view.

Example

The following XML defines a wide view that displays groups of objects. Each new group is started when the value of the [System.ServiceProcess.ServiceController.ServiceType](#)

property changes.

XML

```
<?xml version="1.0" encoding="utf-8" ?>

<Configuration>
  <ViewDefinitions>
    <View>
      <Name>ServiceWideView</Name>
      <ViewSelectedBy>
        <TypeName>System.ServiceProcess.ServiceController</TypeName>
      </ViewSelectedBy>
      <GroupBy>
        <Label>Service Type</Label>
        <PropertyName>ServiceType</PropertyName>
      </GroupBy>
      <WideControl>
        <WideEntries>
          <WideEntry>
            <WideItem>
              <PropertyName>ServiceName</PropertyName>
            </WideItem>
          </WideEntry>
        </WideEntries>
      </WideControl>
    </View>
  </ViewDefinitions>
</Configuration>
```

The following example shows how Windows PowerShell displays the [System.ServiceProcess.ServiceController](#) objects after this format file is loaded.

PowerShell

```
Get-Service f*
```

Output

```
Service Type: Win32OwnProcess
```

```
Fax
```

```
FCSAM
```

```
Service Type: Win32ShareProcess
```

```
fdPHost
```

```
FDResPub
```

```
FontCache
```

```
Service Type: Win32OwnProcess
```

See Also

[Examples of Formatting Files](#)

[Writing a PowerShell Formatting File](#)

List View (Basic)

Article • 03/24/2025

This example shows how to implement a basic list view that displays the `System.ServiceProcess.ServiceController` objects returned by the `Get-Service` cmdlet. For more information about the components of a list view, see [Creating a List View](#).

Load this formatting file

1. Copy the XML from the Example section of this topic into a text file.
2. Save the text file. Be sure to add the `.ps1xml` extension to the file to identify it as a formatting file.
3. Open Windows PowerShell, and run the following command to load the formatting file into the current session: `Update-FormatData -PrependPath PathToFormattingFile.`

⚠️ Warning

This formatting file defines the display of an object that is already defined by a Windows PowerShell formatting file. You must use the `PrependPath` parameter when you run the cmdlet, and you cannot load this formatting file as a module.

Demonstrates

This formatting file demonstrates the following XML elements:

- The `Name` element for the view.
- The `ViewSelectedBy` element that defines what objects are displayed by the view.
- The `ListControl` element that defines what property is displayed by the view.
- The `ListItem` element that defines what is displayed in a row of the list view.
- The `PropertyName` element that defines which property is displayed.

Example

The following XML defines a list view that displays four properties of the [System.ServiceProcess.ServiceController](#) object. In each row, the name of the property is displayed followed by the value of the property.

XML

```
<Configuration>
  <View>
    <Name>System.ServiceProcess.ServiceController</Name>
    <ViewSelectedBy>
      <TypeName>System.ServiceProcess.ServiceController</TypeName>
    </ViewSelectedBy>
    <ListControl>
      <ListEntries>
        <ListEntry>
          <ListItems>
            <ListItem>
              <PropertyName>Name</PropertyName>
            </ListItem>
            <ListItem>
              <PropertyName>DisplayName</PropertyName>
            </ListItem>
            <ListItem>
              <PropertyName>Status</PropertyName>
            </ListItem>
            <ListItem>
              <PropertyName>ServiceType</PropertyName>
            </ListItem>
          </ListItems>
        </ListEntry>
      </ListEntries>
    </ListControl>
  </View>
</Configuration>
```

The following example shows how Windows PowerShell displays the [System.ServiceProcess.ServiceController](#) objects after this format file is loaded.

PowerShell

```
Get-Service f*
```

Output

```
Name      : Fax
DisplayName : Fax
Status    : Stopped
ServiceType : Win32OwnProcess

Name      : FCSAM
```

```
DisplayName : Microsoft Antimalware Service
Status      : Running
ServiceType : Win32OwnProcess

Name       : fdPHost
DisplayName : Function Discovery Provider Host
Status     : Stopped
ServiceType : Win32ShareProcess

Name       : FDResPub
DisplayName : Function Discovery Resource Publication
Status     : Running
ServiceType : Win32ShareProcess

Name       : FontCache
DisplayName : Windows Font Cache Service
Status     : Running
ServiceType : Win32ShareProcess

Name       : FontCache3.0.0.0
DisplayName : Windows Presentation Foundation Font Cache 3.0.0.0
Status     : Stopped
ServiceType : Win32OwnProcess

Name       : FSysAgent
DisplayName : Microsoft Forefront System Agent
Status     : Running
ServiceType : Win32OwnProcess

Name       : FwcAgent
DisplayName : Firewall Client Agent
Status     : Running
ServiceType : Win32OwnProcess
```

See Also

[Examples of Formatting Files](#)

[Writing a PowerShell Formatting File](#)

List View (Labels)

Article • 03/24/2025

This example shows how to implement a list view that displays a custom label for each row of the list. This list view displays the properties of the [System.ServiceProcess.ServiceController](#) object that is returned by the [Get-Service](#) cmdlet. For more information about the components of a list view, see [Creating a List View](#).

Load this formatting file

1. Copy the XML from the Example section of this topic into a text file.
2. Save the text file. Be sure to add the `format.ps1xml` extension to the file to identify it as a formatting file.
3. Open Windows PowerShell, and run the following command to load the formatting file into the current session: `Update-FormatData -PrependPath PathToFormattingFile.`

Warning

This formatting file defines the display of an object that is already defined by a Windows PowerShell formatting file. You must use the `PrependPath` parameter when you run the cmdlet, and you cannot load this formatting file as a module.

Demonstrates

This formatting file demonstrates the following XML elements:

- The [Name](#) element for the view.
- The [ViewSelectedBy](#) element that defines what objects are displayed by the view.
- The [ListControl](#) element that defines what property is displayed by the view.
- The [ListItem](#) element that defines what is displayed in a row of the list view.
- The [Label](#) element that defines what is displayed in a row of the list view.
- The [PropertyName](#) element that defines which property is displayed.

Example

The following XML defines a list view that displays a custom label in each row. In this case, the label includes the property name with each letter capitalized and the word "property". In each row, the name of the property is displayed followed by the value of the property.

XML

```
<Configuration>
  <ViewDefinitions>
    <View>
      <Name>System.ServiceProcess.ServiceController</Name>
      <ViewSelectedBy>
        <TypeName>System.ServiceProcess.ServiceController</TypeName>
      </ViewSelectedBy>
      <ListControl>
        <ListEntries>
          <ListEntry>
            <ListItems>
              <ListItem>
                <Label>NAME property</Label>
                <PropertyName>Name</PropertyName>
              </ListItem>
              <ListItem>
                <Label>DISPLAYNAME property</Label>
                <PropertyName>DisplayName</PropertyName>
              </ListItem>
              <ListItem>
                <Label>STATUS property</Label>
                <PropertyName>Status</PropertyName>
              </ListItem>
              <ListItem>
                <Label>SERVICETYPE property</Label>
                <PropertyName>ServiceType</PropertyName>
              </ListItem>
            </ListItems>
          </ListEntry>
        </ListEntries>
      </ListControl>
    </View>
  </ViewDefinitions>
</Configuration>
```

The following example shows how Windows PowerShell displays the [System.ServiceProcess.ServiceController](#) objects after this format file is loaded.

PowerShell

```
Get-Service f*
```

Output

```
NAME property      : Fax
DISPLAYNAME property : Fax
STATUS property     : Stopped
SERVICETYPE property : Win32OwnProcess

NAME property      : FCSAM
DISPLAYNAME property : Microsoft Antimalware Service
STATUS property     : Running
SERVICETYPE property : Win32OwnProcess

NAME property      : fdPHost
DISPLAYNAME property : Function Discovery Provider Host
STATUS property     : Stopped
SERVICETYPE property : Win32ShareProcess

NAME property      : FDResPub
DISPLAYNAME property : Function Discovery Resource Publication
STATUS property     : Running
SERVICETYPE property : Win32ShareProcess

NAME property      : FontCache
DISPLAYNAME property : Windows Font Cache Service
STATUS property     : Running
SERVICETYPE property : Win32ShareProcess

NAME property      : FontCache3.0.0.0
DISPLAYNAME property : Windows Presentation Foundation Font Cache 3.0.0.0
STATUS property     : Stopped
SERVICETYPE property : Win32OwnProcess

NAME property      : FSysAgent
DISPLAYNAME property : Microsoft Forefront System Agent
STATUS property     : Running
SERVICETYPE property : Win32OwnProcess

NAME property      : FwcAgent
DISPLAYNAME property : Firewall Client Agent
STATUS property     : Running
SERVICETYPE property : Win32OwnProcess
```

See Also

[Examples of Formatting Files](#)

[Writing a PowerShell Formatting File](#)

List View (GroupBy)

Article • 03/24/2025

This example shows how to implement a list view that separates the rows of the list into groups. This list view displays the properties of the [System.ServiceProcess.ServiceController](#) objects returned by the [Get-Service](#) cmdlet. For more information about the components of a list view, see [Creating a List View](#).

Load this formatting file

1. Copy the XML from the Example section of this topic into a text file.
2. Save the text file. Be sure to add the `format.ps1xml` extension to the file to identify it as a formatting file.
3. Open Windows PowerShell, and run the following command to load the formatting file into the current session: `Update-FormatData -PrependPath PathToFormattingFile`.

Warning

This formatting file defines the display of an object that is already defined by a Windows PowerShell formatting file. You must use the `PrependPath` parameter when you run the cmdlet, and you cannot load this formatting file as a module.

Demonstrates

This formatting file demonstrates the following XML elements:

- The [Name](#) element for the view.
- The [ViewSelectedBy](#) element that defines what objects are displayed by the view.
- The [GroupBy](#) element that defines how a new group of objects is displayed.
- The [ListControl](#) element that defines what property is displayed by the view.
- The [ListItem](#) element that defines what is displayed in a row of the list view.
- The [PropertyName](#) element that defines which property is displayed.

Example

The following XML defines a list view that starts a new group whenever the value of the `System.ServiceProcess.ServiceController.Status` property changes. When each group is started, a custom label is displayed that includes the new value of the property.

XML

```
<Configuration>
  <ViewDefinitions>
    <View>
      <Name>System.ServiceProcess.ServiceController</Name>
      <ViewSelectedBy>
        <TypeName>System.ServiceProcess.ServiceController</TypeName>
      </ViewSelectedBy>
      <GroupBy>
        <PropertyName>Status</PropertyName>
        <Label>New Service Status</Label>
      </GroupBy>
      <ListControl>
        <ListEntries>
          <ListEntry>
            <ListItems>
              <ListItem>
                <PropertyName>Name</PropertyName>
              </ListItem>
              <ListItem>
                <PropertyName>DisplayName</PropertyName>
              </ListItem>
              <ListItem>
                <PropertyName>ServiceType</PropertyName>
              </ListItem>
            </ListItems>
          </ListEntry>
        </ListEntries>
      </ListControl>
    </View>
  </ViewDefinitions>
</Configuration>
```

The following example shows how Windows PowerShell displays the `System.ServiceProcess.ServiceController` objects after this format file is loaded. The blank lines added before and after the group label are automatically added by Windows PowerShell.

PowerShell

```
Get-Service f*
```

Output

```
New Service Status: Stopped
```

```
Name      : Fax
DisplayName : Fax
ServiceType : Win32OwnProcess
```

```
New Service Status: Running
```

```
Name      : FCSAM
DisplayName : Microsoft Antimalware Service
ServiceType : Win32OwnProcess
```

```
New Service Status: Stopped
```

```
Name      : fdPHost
DisplayName : Function Discovery Provider Host
ServiceType : Win32ShareProcess
```

```
New Service Status: Running
```

```
Name      : FDResPub
DisplayName : Function Discovery Resource Publication
ServiceType : Win32ShareProcess
```

```
Name      : FontCache
DisplayName : Windows Font Cache Service
ServiceType : Win32ShareProcess
```

```
New Service Status: Stopped
```

```
Name      : FontCache3.0.0.0
DisplayName : Windows Presentation Foundation Font Cache 3.0.0.0
ServiceType : Win32OwnProcess
```

```
New Service Status: Running
```

```
Name      : FSysAgent
DisplayName : Microsoft Forefront System Agent
ServiceType : Win32OwnProcess
```

```
Name      : FwcAgent
DisplayName : Firewall Client Agent
ServiceType : Win32OwnProcess
```

See Also

[Examples of Formatting Files](#)

[Writing a PowerShell Formatting File](#)

Format Schema XML Reference

Article • 09/17/2021

The topics in this section describe the XML elements used by formatting files (Format.ps1xml files). Formatting files define how the .NET object is displayed; they do not change the object itself.

In This Section

[Alignment Element for TableColumnHeader for TableControl \(Format\)](#) Defines how the data in a column header is displayed.

[Alignment Element for TableColumnItem for TableControl \(Format\)](#) Defines how the data in the row is displayed.

[AutoSize Element for TableControl \(Format\)](#) Specifies whether the column size and the number of columns are adjusted based on the size of the data.

[Autosize Element for WideControl \(Format\)](#) Specifies whether the column size and the number of columns are adjusted based on the size of the data.

[ColumnNumber Element for WideControl \(Format\)](#) Specifies the number of columns displayed in the wide view.

[Configuration Element \(Format\)](#) Represents the top-level element of the formatting file.

[Control Element for Controls for Configuration \(Format\)](#) Defines a common control that can be used by all the views of the formatting file and the name that is used to reference the control.

[Control Element for Controls for View \(Format\)](#) Defines a control that can be used by the view and the name that is used to reference the control.

[Controls Element for Configuration \(Format\)](#) Defines the common controls that can be used by all views of the formatting file.

[Controls Element for View \(Format\)](#) Defines the view controls that can be used by a specific view.

[CustomControl Element for Control for Configuration \(Format\)](#) Defines a control. This element is used when defining a common control that can be used by all the views in the formatting file.

[CustomControl Element for Control for Controls for View \(Format\)](#) Defines a control that is used by the view.

[CustomControl Element for GroupBy \(Format\)](#) Defines the custom control that displays the new group.

[CustomControl Element \(Format\)](#) Defines a custom control format for the view.

[CustomControlName Element for ExpressionBinding for Controls for Configuration \(Format\)](#) Specifies the name of a common control. This element is used when defining a common control that can be used by all the views in the formatting file.

[CustomControlName Element for ExpressionBindine for Controls for View \(Format\)](#) Specifies the name of a common control or a view control. This element is used when defining controls that can be used by a view.

[CustomControlName Element of GroupBy \(Format\)](#) Specifies the name of a custom control that is used to display the new group. This element is used when defining a table, list, wide or custom control view.

[CustomEntry Element for CustomControl for Configuration \(Format\)](#) Provides a definition of the common control. This element is used when defining a common control that can be used by all the views in the formatting file.

[CustomEntry Element for CustomEntries for Controls for View \(Format\)](#) Provides a definition of the control. This element is used when defining controls that can be used by a view.

[CustomEntry Element for CustomEntries for View \(Format\)](#) Provides a definition of the custom control view.

[CustomEntry Element for CustomControl for GroupBy \(Format\)](#) Provides a definition of the control. This element is used when defining how a new group of objects is displayed.

[CustomEntries Element for CustomControl for Configuration \(Format\)](#) Provides the definitions of a common control. This element is used when defining a common control that can be used by all the views in the formatting file.

[CustomEntries Element for CustomControl for Controls for View \(Format\)](#) Provides the definitions for the control. This element is used when defining controls that can be used by a view.

[CustomEntries Element for CustomControl for GroupBy \(Format\)](#) Provides the definitions for the control. This element is used when defining how a new group of objects is displayed.

[CustomEntries Element for CustomControl for View \(Format\)](#) Provides the definitions of the custom control view. The custom control view must specify one or more definitions.

[CustomItem Element for CustomEntry for Controls for Configuration](#) Defines what data is displayed by the control and how it is displayed. This element is used when defining a common control that can be used by all the views in the formatting file.

[CustomItem Element for CustomEntry for Controls for View \(Format\)](#) Defines what data is displayed by the control and how it is displayed. This element is used when defining controls that can be used by a view.

[CustomItem Element for CustomEntry for View \(Format\)](#) Defines what data is displayed by the custom control view and how it is displayed. This element is used when defining a custom control view.

[CustomItem Element for CustomEntry for GroupBy \(Format\)](#) Defines what data is displayed by the custom control view and how it is displayed. This element is used when defining how a new group of objects is displayed.

[DefaultSettings Element \(Format\)](#) Defines common settings that apply to all the views of the formatting file. Common settings include displaying errors, wrapping text in tables, defining how collections are expanded, and more.

[DisplayError Element \(Format\)](#) Specifies that the string #ERR is displayed when an error occurs displaying a piece of data.

[EntrySelectedBy Element for CustomEntry for Controls for Configuration \(Format\)](#) Defines the .NET types that use the definition of the common control or the condition that must exist for this control to be used. This element is used when defining a common control that can be used by all the views in the formatting file.

[EntrySelectedBy Element for CustomEntry for Controls for View \(Format\)](#) Defines the .NET types that use this control definition or the condition that must exist for this definition to be used. This element is used when defining controls that can be used by a view.

[EntrySelectedBy Element for CustomEntry for View \(Format\)](#) Defines the .NET types that use this custom entry or the condition that must exist for this entry to be used.

[EntrySelectedBy Element for EnumerableExpansion \(Format\)](#) Defines the .NET types that use this definition or the condition that must exist for this definition to be used.

[EntrySelectedBy Element for CustomEntry for GroupBy \(Format\)](#) Defines the .NET types that use this control definition or the condition that must exist for this definition to be used. This element is used when defining how a new group of objects is displayed.

[EntrySelectedBy Element for ListEntry for ListControl \(Format\)](#) Defines the .NET types that use this list view definition or the condition that must exist for this definition to be used. In most cases only one definition is needed for a list view. However, you can provide multiple definitions for the list view if you want to use the same list view to display different data for different objects.

[EntrySelectedBy Element for TableRowEntry \(Format\)](#) Defines the .NET types whose property values are displayed in the row.

[EntrySelectedBy Element for WideEntry \(Format\)](#) Defines the .NET types that use this definition of the wide view or the condition that must exist for this definition to be used.

[EnumerableExpansion Element \(Format\)](#) Defines how specific .NET collection objects are expanded when they are displayed in a view.

[EnumerableExpansions Element \(Format\)](#) Defines how .NET collection objects are expanded when they are displayed in a view.

[EnumerateCollection Element for ExpressionBinding for Controls for Configuration \(Format\)](#) Specified that the elements of collections are displayed by the control. This element is used when defining a common control that can be used by all the views in the formatting file.

[EnumerateCollection Element for ExpressionBinding for Controls for View \(Format\)](#) Specified that the elements of collections are displayed. This element is used when defining controls that can be used by a view.

[EnumerateCollection Element for Expression Binding for CustomControl for View \(Format\)](#) Specifies that the elements of collections are displayed. This element is used when defining a custom control view.

[EnumerateCollection Element for ExpressionBinding for GroupBy \(Format\)](#) Specifies that the elements of collections are displayed. This element is used when defining how a new group of objects is displayed.

[Expand Element \(Format\)](#) Specifies how the collection object is expanded for this definition.

[ExpressionBinding Element for CustomItem for Controls for Configuration \(Format\)](#) Defines the data that is displayed by the control. This element is used when defining a common control that can be used by all the views in the formatting file.

[ExpressionBinding Element for CustomItem for Controls for View \(Format\)](#) Defines the data that is displayed by the control. This element is used when defining controls that can be used by a view.

[ExpressionBinding Element for CustomItem for CustomControl for View \(Format\)](#) Defines the data that is displayed by the control. This element is used when defining a custom control view.

[ExpressionBinding Element for CustomItem for GroupBy \(Format\)](#) Defines the data that is displayed by the control. This element is used when defining how a new group of objects is displayed.

[FirstLineHanging Element for Frame for Controls for Configuration \(Format\)](#) Specifies how many characters the first line of data is shifted to the left. This element is used when defining a common control that can be used by all the views in the formatting file.

[FirstLineHanging Element of Frame of Controls of View \(Format\)](#) Specifies how many characters the first line of data is shifted to the left. This element is used when defining controls that can be used by a view.

[FirstLineHanging Element for Frame for CustomControl for View \(Format\)](#) Specifies how many characters the first line of data is shifted to the left. This element is used when defining a custom control view.

[FirstLineHanging Element for Frame for GroupBy \(Format\)](#) Specifies how many characters the first line of data is shifted to the left. This element is used when defining how a new group of objects is displayed.

[FirstLineIndent Element for Frame for Controls for Configuration \(Format\)](#) Specifies how many characters the first line of data is shifted to the right. This element is used when defining a common control that can be used by all the views in the formatting file.

[FirstLineIndent Element of Frame of Controls of View \(Format\)](#) Specifies how many characters the first line of data is shifted to the right. This element is used when defining controls that can be used by a view.

[FirstLineIndent Element](#) Specifies how many characters the first line of data is shifted to the right. This element is used when defining a custom control view.

[FirstLineIndent Element for Frame for GroupBy \(Format\)](#) Specifies how many characters the first line of data is shifted to the right. This element is used when defining how a new group of objects is displayed.

[FormatString Element for ListItem \(Format\)](#) Specifies a format pattern that defines how the property or script value is displayed.

[FormatString Element for TableColumnItem \(Format\)](#) Specifies a format pattern that defines how the property or script value of the table is displayed.

[FormatString Element for WideItem for WideControl \(Format\)](#) Specifies a format pattern that defines how the property or script value is displayed in the view.

[Frame Element for CustomItem for Controls for Configuration \(Format\)](#) Defines how the data is displayed, such as shifting the data to the left or right. This element is used when defining a common control that can be used by all the views in the formatting file.

[Frame Element for CustomItem for Controls for View \(Format\)](#) Defines how the data is displayed, such as shifting the data to the left or right. This element is used when defining controls that can be used by a view.

[Frame Element for CustomItem for CustomControl for View \(Format\)](#) Defines how the data is displayed, such as shifting the data to the left or right. This element is used when defining a custom control view.

[Frame Element for CustomItem for GroupBy \(Format\)](#) Defines how the data is displayed, such as shifting the data to the left or right. This element is used when defining how a new group of objects is displayed.

[GroupBy Element for View \(Format\)](#) Defines how Windows PowerShell displays a new group of objects.

[HideTableHeaders Element \(Format\)](#) Specifies that the headers of the table are not displayed.

[ItemSelectionCondition Element for ExpressionBinding for Controls for Configuration \(Format\)](#) Defines the condition that must exist for this control to be used. This element is used when defining a common control that can be used by all the views in the formatting file.

[ItemSelectionCondition Element of ExpressionBinding for Controls for View \(Format\)](#) Defines the condition that must exist for this control to be used. This element is used when defining controls that can be used by a view.

[ItemSelectionCondition Element for Expression Binding for CustomControl for View \(Format\)](#) Defines the condition that must exist for this control to be used. There is no limit to the number of selection conditions that can be specified for a control item. This element is used when defining a custom control view.

[ItemSelectionCondition Element for ExpressionBinding for GroupBy \(Format\)](#) Defines the condition that must exist for this control to be used. There is no limit to the number of selection conditions that can be specified for a control item. This element is used when defining how a new group of objects is displayed.

[ItemSelectionCondition Element for ListItem \(Format\)](#) Defines the condition that must exist for this list item to be used.

[Label Element for ListItem for ListControl\(Format\)](#) Specifies the label for the property or script value in the row.

[Label Element for GroupBy \(Format\)](#) Specifies a label that is displayed when a new group is encountered.

[Label Element for TableColumnHeader \(Format\)](#) Defines the label that is displayed at the top of a column.

[LeftIndent Element for Frame for Controls for Configuration \(Format\)](#) Specifies how many characters the data is shifted away from the left margin. This element is used when defining a common control that can be used by all the views in the formatting file.

[LeftIndent Element of Frame of Controls of View \(Format\)](#) Specifies how many characters the data is shifted away from the left margin. This element is used when defining controls that can be used by a view.

[LeftIndent Element for Frame for CustomControl for View \(Format\)](#) Specifies how many characters the data is shifted away from the left margin. This element is used when defining a custom control view.

[LeftIndent Element for Frame for GroupBy \(Format\)](#) Specifies how many characters the data is shifted away from the left margin. This element is used when defining how a new group of objects is displayed.

[ListControl Element \(Format\)](#) Defines a list format for the view.

[ListEntry Element \(Format\)](#) Provides a definition of the list view.

[ListEntries Element \(Format\)](#) Defines how the rows of the list view are displayed.

[ListItem Element \(Format\)](#) Defines the property or script whose value is displayed in a row of the list view.

[ListItems Element \(Format\)](#) Defines the properties and scripts that are displayed in the list view.

[Name Element for Control for Controls for Configuration \(Format\)](#) Specifies the name of the control. This element is used when defining a common control that can be used by all the views in the formatting file.

[Name Element for SelectionSet \(Format\)](#) Specifies the name used to reference the selection set.

[Name Element for View \(Format\)](#) Specifies the name that is used to identify the view.

[NewLine Element for CustomItem for Controls for Configuration \(Format\)](#) Adds a blank line to the display of the control. This element is used when defining a common control that can be used by all the views in the formatting file.

[NewLine Element for CustomItem for Controls for View \(Format\)](#) Adds a blank line to the display of the control. This element is used when defining controls that can be used by a view.

[NewLine Element for CustomItem for CustomControl for View \(Format\)](#) Adds a blank line to the display of the control. This element is used when defining a custom control view.

[NewLine Element for CustomItem for GroupBy \(Format\)](#) Adds a blank line to the display of the control. This element is used when defining how a new group of objects is displayed.

[PropertyName Element for ExpressionBinding for Controls for Configuration \(Format\)](#) Specifies the .NET property whose value is displayed by the common control. This element is used when defining a common control that can be used by all the views in the formatting file.

[PropertyName Element for ExpressionBinding for Controls for View \(Format\)](#) Specifies the .NET property whose value is displayed by the control. This element is used when defining controls that can be used by a view.

[PropertyName Element for ExpressionBinding for CustomControl for View \(Format\)](#) Specifies the .NET property whose value is displayed by the control. This element is used when defining a custom control view

[PropertyName Element for ExpressionBinding for GroupBy \(Format\)](#) Specifies the .NET property whose value is displayed by the control. This element is used when defining how a new group of objects is displayed.

[PropertyName Element for GroupBy \(Format\)](#) Specifies the .NET property that starts a new group whenever its value changes.

[PropertyName Element for ItemSelectionCondition for Controls for Configuration \(Format\)](#) Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the control is used. This element is used when defining a common control that can be used by all the views in the formatting file.

[PropertyName Element for ItemSelectionCondition for Controls for View \(Format\)](#)

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the control is used. This element is used when defining controls that can be used by a view.

[PropertyName Element for ItemSelectionCondition for CustomControl for View \(Format\)](#)

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the control is used. This element is used when defining a custom control view.

[PropertyName Element for ItemSelectionCondition for GroupBy \(Format\)](#) Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the control is used. This element is used when defining how a new group of objects is displayed.

[PropertyName Element for ItemSelectionCondition for ListItem \(Format\)](#) Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the view is used. This element is used when defining a list view.

[PropertyName Element for ListItem for ListControl \(Format\)](#) Specifies the .NET property whose value is displayed in the list.

[PropertyName Element for SelectionCondition for EntrySelectedBy for ListEntry \(Format\)](#) Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the entry is used. This element is used when defining a common control that can be used by all the views in the formatting file.

[PropertyName Element for SelectionCondition for Controls for View \(Format\)](#) Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the entry is used. This element is used when defining controls that can be used by a view.

[PropertyName Element for SelectionCondition for CustomControl for View \(Format\)](#)

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the definition is used. This element is used when defining a custom control view.

[PropertyName Element for SelectionCondition for EntrySelectedBy for](#)

[EnumerableExpansion \(Format\)](#) Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the definition is used.

[PropertyName Element for SelectionCondition for GroupBy \(Format\)](#) Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the definition is used. This element is used when defining how a new group of objects is displayed.

[PropertyName Element for SelectionCondition for EntrySelectedBy for ListEntry \(Format\)](#) Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the list entry is used.

[PropertyName Element for SelectionCondition for EntrySelectedBy for TableRowEntry \(Format\)](#) Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the table entry is used.

[PropertyName Element for SelectionCondition for EntrySelectedBy for WideEntry \(Format\)](#) Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the definition is used.

[PropertyName Element for TableColumnItem \(Format\)](#) Specifies the property whose value is displayed in the column of the row.

[PropertyName Element for Wideltem \(Format\)](#) Specifies the property of the object whose value is displayed in the wide view.

[RightIndent Element for Frame for Controls for Configuration \(Format\)](#) Specifies how many characters the data is shifted away from the right margin. This element is used when defining a common control that can be used by all the views in the formatting file.

[RightIndent Element of Frame of Controls of View \(Format\)](#) Specifies how many characters the data is shifted away from the right margin. This element is used when defining controls that can be used by a view.

[RightIndent Element](#) Specifies how many characters the data is shifted away from the right margin. This element is used when defining a custom control view.

[RightIndent Element for Frame for GroupBy \(Format\)](#) Specifies how many characters the data is shifted away from the right margin. This element is used when defining how a new group of objects is displayed.

[ScriptBlock Element for ExpressionBinding for Controls for Configuration \(Format\)](#) Specifies the script whose value is displayed by the common control. This element is used when defining a common control that can be used by all the views in the formatting file.

[ScriptBlock Element for ExpressionBinding for Controls for View \(Format\)](#) Specifies the script whose value is displayed by the control. This element is used when defining controls that can be used by a view.

[ScriptBlock Element for ExpressionBinding for CustomCustomControl for View \(Format\)](#) Specifies the script whose value is displayed by the control. This element is used when defining a custom control view.

[ScriptBlock Element for ExpressionBinding for GroupBy \(Format\)](#) Specifies the script whose value is displayed by the control. This element is used when defining how a new group of objects is displayed.

[ScriptBlock Element for GroupBy \(Format\)](#) Specifies the script that starts a new group whenever its value changes.

[ScriptBlock Element for ItemSelectionCondition for Controls for Configuration \(Format\)](#) Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the control is used. This element is used when defining a common control that can be used by all the views in the formatting file.

[ScriptBlock Element for ItemSelectionCondition for Controls for View \(Format\)](#) Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the control is used. This element is used when defining controls that can be used by a view.

[ScriptBlock Element for ItemSelectionCondition for CustomControl for View \(Format\)](#) Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the control is used. This element is used when defining a custom control view.

[ScriptBlock Element for ItemSelectionCondition for GroupBy \(Format\)](#) Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the control is used. This element is used when defining how a new group of objects is displayed.

[ScriptBlock Element for ItemSelectionCondition for ListControl \(Format\)](#) Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the list item is used. This element is used when defining a list view.

[ScriptBlock Element for ListItem \(Format\)](#) Specifies the script whose value is displayed in the row of the list.

[ScriptBlock Element for SelectionCondition for Controls for Configuration \(Format\)](#) Specifies the script that triggers the condition. When this script is evaluated to `true`, the

condition is met, and the definition is used. This element is used when defining a common control that can be used by all the views in the formatting file.

[ScriptBlock Element for SelectionCondition for Controls for View \(Format\)](#) Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the definition is used. This element is used when defining controls that can be used by a view.

[ScriptBlock Element for SelectionCondition for CustomControl for View \(Format\)](#) Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the definition is used. This element is used when defining a custom control view.

[ScriptBlock Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion \(Format\)](#) Specifies the script that triggers the condition.

[ScriptBlock Element for SelectionCondition for GroupBy \(Format\)](#) Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the definition is used. This element is used when defining how a new group of objects is displayed.

[ScriptBlock Element for SelectionCondition for EntrySelectedBy for ListEntry \(Format\)](#) Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the list entry is used.

[ScriptBlock Element for SelectionCondition for EntrySelectedBy for TableRowEntry \(Format\)](#) Specifies the script block that triggers the condition. When this script is evaluated to `true`, the condition is met, and the table entry is used.

[ScriptBlock Element for SelectionCondition for EntrySelectedBy for WideEntry \(Format\)](#) Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the wide entry definition is used.

[ScriptBlock Element for TableColumnItem \(Format\)](#) Specifies the script whose value is displayed in the column of the row.

[ScriptBlock Element for Wideltem \(Format\)](#) Specifies the script whose value is displayed in the wide view.

[SelectionCondition Element for EntrySelectedBy for CustomEntry for Configuration \(Format\)](#) Defines a condition that must exist for a common control definition to be used. This element is used when defining a common control that can be used by all the views in the formatting file.

[SelectionCondition Element for EntrySelectedBy for Controls for View \(Format\)](#) Defines a condition that must exist for the control definition to be used. This element is used when defining controls that can be used by a view.

[SelectionCondition Element for EntrySelectedBy for CustomControl for View \(Format\)](#) Defines a condition that must exist for a control definition to be used. This element is used when defining a custom control view.

[SelectionCondition Element for EntrySelectedBy for EnumerableExpansion \(Format\)](#) Defines the condition that must exist to expand the collection objects of this definition.

[SelectionCondition Element for EntrySelectedBy for GroupBy \(Format\)](#) Defines a condition that must exist for a control definition to be used. This element is used when defining how a new group of objects is displayed.

[SelectionCondition Element for EntrySelectedBy for ListEntry \(Format\)](#) Defines the condition that must exist to use this definition of the list view. There is no limit to the number of selection conditions that can be specified for a list definition.

[SelectionCondition Element for EntrySelectedBy for TableRowEntry \(Format\)](#) Defines the condition that must exist to use for this definition of the table view. There is no limit to the number of selection conditions that can be specified for a table definition.

[SelectionCondition Element for EntrySelectedBy for WideEntry \(Format\)](#) Defines the condition that must exist for this definition to be used. There is no limit to the number of selection conditions that can be specified for a wide entry definition.

[SelectionSet Element \(Format\)](#) Defines a set of .NET objects that can be referenced by the name of the set.

[SelectionSetName Element for EntrySelectedBy for Controls for Configuration \(Format\)](#) Specifies a set of .NET types that use this definition of the control. This element is used when defining a common control that can be used by all the views in the formatting file.

[SelectionSetName Element for EntrySelectedBy for Controls for View \(Format\)](#) Specifies a set of .NET types that use this definition of the control. This element is used when defining controls that can be used by a view.

[SelectionSetName Element for EntrySelectedBy for CustomEntry \(Format\)](#) Specifies a set of .NET objects for the list entry. There is no limit to the number of selection sets that can be specified for an entry.

[SelectionSetName Element for EntrySelectedBy for EnumerableExpansion \(Format\)](#) Specifies the set of .NET types that are expanded by this definition.

[SelectionSetName Element for EntrySelectedBy for GroupBy \(Format\)](#) Specifies a set of .NET objects for the list entry. There is no limit to the number of selection sets that can be specified for an entry. This element is used when defining how a new group of objects is displayed.

[SelectionSetName Element for EntrySelectedBy for ListEntry \(Format\)](#) Specifies a set of .NET objects for the list entry. There is no limit to the number of selection sets that can be specified for an entry.

[SelectionSetName Element for EntrySelectedBy for TableRowEntry \(Format\)](#) Specifies a set of .NET types the use this entry of the table view. There is no limit to the number of selection sets that can be specified for an entry.

[SelectionSetName Element for EntrySelectedBy for WideEntry \(Format\)](#) Specifies a set of .NET objects for the definition. The definition is used whenever one of these objects is displayed.

[SelectionSetName Element for SelectionCondition for Controls for Configuration \(Format\)](#) Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met, and the object is displayed by using this control. This element is used when defining a common control that can be used by all the views in the formatting file.

[SelectionSetName Element for SelectionCondition for Controls for View \(Format\)](#) Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met and the object is displayed using this control. This element is used when defining controls that can be used by a view.

[EntrySelectedBy Element for CustomEntry for View \(Format\)](#) Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met and the object is displayed using this control. This element is used when defining a custom control view.

[SelectionSetName Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion \(Format\)](#) Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met.

[SelectionSetName Element for SelectionCondition for GroupBy \(Format\)](#) Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met, and the object is displayed by using this control. This element is used when defining how a new group of objects is displayed.

[SelectionSetName Element for SelectionCondition for EntrySelectedBy for ListEntry \(Format\)](#) Specifies the set of .NET types that trigger the condition. When any of the

types in this set are present, the condition is met, and the object is displayed by using this definition of the list view.

[SelectionSetName Element for SelectionCondition for EntrySelectedBy for TableRowEntry \(Format\)](#) Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met, and the object is displayed by using this definition of the table view.

[SelectionSetName Element for SelectionCondition for EntrySelectedBy for WideEntry \(Format\)](#) Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met, and the object is displayed by using this definition of the wide view.

[SelectionSetName Element for ViewSelectedBy \(Format\)](#) Specifies a set of .NET objects that are displayed by the view.

[SelectionSets Element \(Format\)](#) Defines the sets of .NET objects that can be used by individual format views.

[ShowError Element \(Format\)](#) Specifies that the full error record is displayed when an error occurs while displaying a piece of data.

[TableColumnHeader Element for TableHeaders for TableControl \(Format\)](#) Defines the label, the width of the column, and the alignment of the label for a column of the table.

[TableColumnItem Element \(Format\)](#) Defines the property or script whose value is displayed in the column of the row.

[TableColumnItems Element \(Format\)](#) Defines the properties or scripts whose values are displayed in the row.

[TableControl Element \(Format\)](#) Defines a table format for a view.

[TableHeaders Element \(Format\)](#) Defines the headers for the columns of a table.

[TableRowEntries Element \(Format\)](#) Defines the rows of the table.

[TableRowEntry Element \(Format\)](#) Defines the data that is displayed in a row of the table.

[Text Element for CustomItem for Controls for Configuration \(Format\)](#) Specifies text that is added to the data that is displayed by the control, such as a label, brackets to enclose the data, and spaces to indent the data. This element is used when defining a common control that can be used by all the views in the formatting file.

[Text Element for CustomItem for Controls for View \(Format\)](#) Specifies text that is added to the data that is displayed by the control, such as a label, brackets to enclose the data,

and spaces to indent the data. This element is used when defining controls that can be used by a view.

[Text Element for CustomItem \(Format\)](#) Specifies text that is added to the data that is displayed by the control, such as a label, brackets to enclose the data, and spaces to indent the data. This element is used when defining a custom control view.

[Text Element for CustomItem for GroupBy \(Format\)](#) Specifies text that is added to the data that is displayed by the control, such as a label, brackets to enclose the data, and spaces to indent the data. This element is used when defining how a new group of objects is displayed.

[TypeName Element for EntrySelectedBy for Controls for Configuration \(Format\)](#) Specifies a .NET type that uses this definition of the control. This element is used when defining a common control that can be used by all the views in the formatting file.

[TypeName Element for EntrySelectedBy for Controls for View \(Format\)](#) Specifies a .NET type that uses this definition of the control. This element is used when defining controls that can be used by a view.

[TypeName Element for EntrySelectedBy for CustomEntry for View \(Format\)](#) Specifies a .NET type that uses this definition of the custom control view. There is no limit to the number of types that can be specified for a definition.

[TypeName Element for EntrySelectedBy for EnumerableExpansion \(Format\)](#) Specifies a .NET type that is expanded by this definition. This element is used when defining a default settings.

[TypeName Element for EntrySelectedBy for GroupBy \(Format\)](#) Specifies a .NET type that uses this definition of the custom control. This element is used when defining how a new group of objects is displayed.

[TypeName Element for EntrySelectedBy for ListControl \(Format\)](#) Specifies a .NET type that uses this entry of the list view. There is no limit to the number of types that can be specified for a list entry.

[TypeName Element for EntrySelectedBy for TableRowEntry \(Format\)](#) Specifies a .NET type that uses this entry of the table view. There is no limit to the number of types that can be specified for a table entry.

[TypeName Element for EntrySelectedBy for WideEntry \(Format\)](#) Specifies a .NET type for the definition. The definition is used whenever this object is displayed.

[TypeName Element for SelectionCondition for Controls for Configuration \(Format\)](#) Specifies a .NET type that triggers the condition. This element is used when defining a

common control that can be used by all the views in the formatting file.

[TypeName Element for SelectionCondition for Controls for View \(Format\)](#) Specifies a .NET type that triggers the condition. This element is used when defining controls that can be used by a view.

[TypeName Element for SelectionCondition for CustomControl for View \(Format\)](#)

Specifies a .NET type that triggers the condition. This element is used when defining a custom control view.

[TypeName Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion \(Format\)](#) Specifies a .NET type that triggers the condition.

[TypeName Element for SelectionCondition for GroupBy \(Format\)](#) Specifies a .NET type that triggers the condition. This element is used when defining how a new group of objects is displayed.

[TypeName Element for SelectionCondition for EntrySelectedBy for ListControl \(Format\)](#) Specifies a .NET type that triggers the condition. When this type is present, the list entry is used.

[TypeName Element for SelectionCondition for EntrySelectedBy for TableRowEntry \(Format\)](#) Specifies a .NET type that triggers the condition. When this type is present, the condition is met, and the table row is used.

[TypeName Element for SelectionCondition for EntrySelectedBy for WideEntry \(Format\)](#) Specifies a .NET type that triggers the condition. When this type is present, the definition is used.

[TypeName Element for Types \(Format\)](#) Specifies the .NET type of an object that belongs to the selection set.

[TypeName Element for ViewSelectedBy \(Format\)](#) Specifies a .NET object that is displayed by the view.

[Types Element \(Format\)](#) Defines the .NET objects that are in the selection set.

[View Element \(Format\)](#) Defines a view that is used to display one or more .NET objects.

[ViewDefinitions Element \(Format\)](#) Defines the views used to display objects.

[ViewSelectedBy Element \(Format\)](#) Defines the .NET objects that are displayed by the view.

[WideControl Element \(Format\)](#) Defines a wide (single value) list format for the view. This view displays a single property value or script value for each object.

[WideEntries Element \(Format\)](#) Provides the definitions of the wide view. The wide view must specify one or more definitions.

[WideEntry Element \(Format\)](#) Provides a definition of the wide view.

[WideItem Element \(Format\)](#) Defines the property or script whose value is displayed.

[Width Element \(Format\)](#) Defines the width (in characters) of a column.

[Wrap Element \(Format\)](#) Specifies that text that exceeds the column width is displayed on the next line.

[WrapTables Element \(Format\)](#) Specifies that data in a table cell is moved to the next line if the data is longer than the width of the column.

See Also

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Configuration Element

Article • 09/17/2021

Represents the top-level element of a formatting file.

Schema

- Configuration Element

Syntax

XML

```
<Configuration>
  <DefaultSettings>...</DefaultSettings>
  <SelectionSets>...</SelectionSets>
  <Controls>...</Controls>
  <ViewDefinitions>...</ViewDefinitions>
</Configuration>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `Configuration` element. This element must be the root element for each formatting file, and this element must contain at least one child element.

Attributes

None.

Child Elements

[] Expand table

Element	Description
Controls Element for Configuration	Optional element.

Element	Description
	Defines the common controls that can be used by all views of the formatting file.
DefaultSettings Element	Optional element. Defines common settings that apply to all the views of the formatting file.
SelectionSets Element Format	Optional element. Defines the common sets of .NET objects that can be used by all views of the formatting file.
ViewDefinitions Element	Optional element. Defines the views used to display objects.

Parent Elements

None.

Remarks

Formatting files define how objects are displayed. In most cases, this root element contains a [ViewDefinitions](#) element that defines the table, list, and wide views of the formatting file. In addition to the view definitions, the formatting file can define common selection sets, settings, and controls that those views can use.

See Also

[Controls Element for Configuration](#)

[DefaultSettings Element](#)

[SelectionSets Element](#)

[ViewDefinitions Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Controls Element for Configuration

Article • 09/17/2021

Defines the common controls that can be used by all views of the formatting file.

Schema

- Configuration Element
- Controls Element

Syntax

XML

```
<Controls>
  <Control>...</Control>
</Controls>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `Controls` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
Control Element for Controls for Configuration	Required element. Defines a common control that can be used by all views of the formatting file.

Parent Elements

Element	Description
Configuration Element	Represents the top-level element of a formatting file.

Remarks

You can create any number of common controls. For each control, you must specify the name that is used to reference the control and the components of the control.

See Also

[Configuration Element](#)

[Control Element for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Control Element for Controls for Configuration

Article • 09/17/2021

Defines a common control that can be used by all the views of the formatting file and the name that is used to reference the control.

Schema

- Configuration Element
- Controls Element
- Control Element

Syntax

XML

```
<Control>
  <Name>NameOfControl</Name>
  <CustomControl>...</CustomControl>
</Control>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element for the `Control` element. You must specify only one of each child element.

Attributes

None.

Child Elements

[] Expand table

Element	Description
CustomControl Element for Control for Controls for Configuration	Required element.

Element	Description
Name Element for Control for Configuration	<p>Required element.</p> <p>Specifies the name used to reference the control.</p>

Parent Elements

[\[+\] Expand table](#)

Element	Description
Controls Element of Configuration	Defines the common controls that can be used by all views of the formatting file or by other controls.

Remarks

The name given to this control can be referenced in the following elements:

- [ExpressionBinding Element for CustomItem](#)
- [GroupBy Element for View](#)

See Also

[Controls Element of Configuration](#)

[CustomControl element for Control for Configuration](#)

[ExpressionBinding Element for CustomItem](#)

[GroupBy Element for View\(Format\)](#)

[Name Element for Control for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

CustomControl Element for Control for Controls for Configuration

Article • 09/17/2021

Defines a control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element

Syntax

XML

```
<CustomControl>
  <CustomEntries>...</CustomEntries>
</CustomControl>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `CustomControl` element. This element must have at least one child element. There is no maximum limit to the number of child elements that can be specified.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
CustomEntries Element for CustomControl for Configuration	Required element. Provides the definitions of a control.

Parent Elements

[\[+\] Expand table](#)

Element	Description
Control Element for Controls for Configuration	Defines a common control that can be used by all the views of the formatting file and the name that is used to reference the control.

Remarks

See Also

[Control Element for Controls for Configuration](#)

[CustomEntries Element for CustomControl for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

CustomEntries Element for CustomControl for Controls for Configuration

Article • 09/17/2021

Provides the definitions of a common control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element

Syntax

XML

```
<CustomEntries>
  <CustomEntry>...</CustomEntry>
</CustomEntries>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomEntries` element. You must specify one or more child elements.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
CustomEntry Element for CustomControl for Controls for Configuration	Provides a definition of the common control.

Parent Elements

[+] Expand table

Element	Description
CustomControl Element for Control for Configuration	Defines a common control.

Remarks

In most cases, a control has only one definition, which is defined in a single `CustomEntry` element. However it is possible to have multiple definitions if you want to use the same control to display different .NET objects. In those cases, you can define a `CustomEntry` element for each object or set of objects.

See Also

[CustomControl Element for Control for Configuration](#)

[CustomEntry Element for CustomControl for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomEntry Element for CustomControl for Controls for Configuration

Article • 09/17/2021

Provides a definition of the common control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element

Syntax

XML

```
<CustomEntry>
  <EntrySelectedBy>...</EntrySelectedBy>
  <CustomItem>...</CustomItem>
</CustomEntry>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomEntry` element. You must specify the items displayed by the definition.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for Controls for Configuration	Optional element. Defines the .NET types that use the definition of the common control or the condition that must exist for this control to be used.
CustomItem Element for CustomEntry for Controls for Configuration	Required element. Defines what data is displayed by the control and how it is displayed.

Parent Elements

[+] Expand table

Element	Description
CustomEntries Element for CustomControl for Configuration	Provides the definitions of the common control.

Remarks

In most cases, only one definition is required for each common custom control, but it is possible to have multiple definitions if you want to use the same control to display different .NET objects. In those cases, you can provide a separate definition for each object or set of objects.

See Also

[CustomEntries Element for CustomControl for Configuration](#)

[CustomItem Element for CustomEntry for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub



PowerShell feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomItem Element for CustomEntry for Controls for Configuration

Article • 09/17/2021

Defines what data is displayed by the control and how it is displayed. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element

Syntax

XML

```
<CustomItem>
  <ExpressionBinding>...</ExpressionBinding>
  <NewLine/>
  <Text>TextToDisplay</Text>
  <Frame>...</Frame>
</CustomItem>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomItem` element. For more information, see Remarks.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
ExpressionBinding Element for CustomItem for Controls for Configuration	Optional element. Defines the data that is displayed by the control.
Frame Element for CustomItem for Controls for Configuration	Optional element. Defines how the data is displayed, such as shifting the data to the left or right.
NewLine Element for CustomItem for Controls for Configuration	Optional element. Adds a blank line to the display of the control.
Text Element for CustomItem for Controls for Configuration	Optional element. Adds text, such as parentheses or brackets, to the display of the control.

Parent Elements

[+] Expand table

Element	Description
CustomEntry Element for CustomControl for Controls for Configuration	Provides a definition of the control.

Remarks

When specifying the child elements of the `CustomItem` element, keep the following in mind:

- The child elements must be added in the following sequence: `ExpressionBinding`, `NewLine`, `Text`, and `Frame`.
- There is no maximum limit to the number of sequences that you can specify.
- In each sequence, there is no maximum limit to the number of `ExpressionBinding` elements that you can use.

See Also

[ExpressionBinding Element for CustomItem for Controls for Configuration](#)

[Frame Element for CustomItem for Controls for Configuration](#)

[NewLine Element for CustomItem for Controls for Configuration](#)

[Text Element for CustomItem for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ExpressionBinding Element for CustomItem for Controls for Configuration

Article • 09/17/2021

Defines the data that is displayed by the control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element

Syntax

XML

```
<ExpressionBinding>
  <CustomControl>...</CustomControl>
  <CustomControlName>NameofCommonCustomControl</CustomControlName>
  <EnumerateCollection/>
  <ItemSelectionCondition>...</ItemSelectionCondition>
  <PropertyName>Nameof.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</ExpressionBinding>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ExpressionBinding` element.

Attributes

None.

Child Elements

[] Expand table

Element	Description
CustomControl Element	Optional element. Defines a control that is used by this control.
CustomControlName Element for ExpressionBinding for Controls for Configuration	Optional element. Specifies the name of a common control or a view control.
EnumerateCollection Element for ExpressionBinding for Controls for Configuration	Optional element. Specified that the elements of collections are displayed by the control.
ItemSelectionCondition Element for ExpressionBinding for Controls for Configuration	Optional element. Defines the condition that must exist for this common control to be used.
PropertyName Element for ExpressionBinding for Controls for Configuration	Optional element. Specifies the .NET property whose value is displayed by the common control.
ScriptBlock Element for ExpressionBinding for Controls for Configuration	Optional element. Specifies the script whose value is displayed by the common control.

Parent Elements

[] Expand table

Element	Description
CustomItem Element for CustomEntry for Controls for Configuration	Defines what data is displayed by the custom control view and how it is displayed.

Remarks

See Also

[CustomItem Element for CustomEntry for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

CustomControlName Element for ExpressionBinding for Controls for Configuration

Article • 09/17/2021

Specifies the name of a common control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- CustomControlName Element

Syntax

XML

```
<CustomControlName>NameofCustomControl</CustomControlName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomControlName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for Controls for Configuration	Defines the data that is displayed by the control.

Text Value

Specify the name of the control.

Remarks

You can create common controls that can be used by all the views of a formatting file, and you can create view controls that can be used by a specific view. The following elements specify the names of these controls:

- [Name Element for Control for Controls for Configuration](#)
- [Name Element for Control for Controls for View](#)

See Also

[Name Element for Control for Controls for Configuration](#)

[Name Element for Control for Controls for View](#)

[ExpressionBinding Element for CustomItem for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

EnumerateCollection Element for ExpressionBinding for Controls for Configuration

Article • 09/17/2021

Specified that the elements of collections are displayed by the control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- EnumerateCollection Element

Syntax

XML

```
<EnumerateCollection/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `EnumerateCollection` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for Controls for Configuration	Defines the data that is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ItemSelectionCondition Element for ExpressionBinding for Controls for Configuration

Article • 09/17/2021

Defines the condition that must exist for this control to be used. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ItemSelectionCondition Element

Syntax

XML

```
<ItemSelectionCondition>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</ItemSelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ItemSelectionCondition` element.

Attributes

None.

Child Elements

[Expand table](#)

Element	Description
PropertyName Element for ItemSelectionCondition for Controls for Configuration	Optional element. Specifies the .NET property that triggers the condition.
ScriptBlock Element for ItemSelectionCondition for Controls for Configuration	Optional element. Specifies the script that triggers the condition.

Parent Elements

[Expand table](#)

Element	Description
ExpressionBinding Element for CustomItem for Controls for Configuration	Defines the data that is displayed by the control.

Remarks

You can specify one property name or a script for this condition but cannot specify both.

See Also

[PropertyName Element for ItemSelectionCondition for Controls for Configuration](#)

[ScriptBlock Element for ItemSelectionCondition for Controls for Configuration](#)

[ExpressionBinding Element for CustomItem for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub



PowerShell feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PropertyName Element for ItemSelectionCondition for Controls for Configuration

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the control is used. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ItemSelectionCondition Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] Expand table

Element	Description
ItemSelectionCondition Element for ExpressionBinding for Controls for Configuration	Defines the condition that must exist for this control to be used.

Text Value

Specify the name of the .NET property that triggers the condition.

Remarks

If this element is used, you cannot specify the [ScriptBlock](#) element when defining the selection condition.

See Also

[ScriptBlock Element for ItemSelectionCondition for Controls for Configuration](#)

[ItemSelectionCondition Element for ExpressionBinding for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

ScriptBlock Element for ItemSelectionCondition for Controls for Configuration

Article • 09/17/2021

Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the control is used. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ItemSelectionCondition Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ItemSelectionCondition Element for ExpressionBinding for Controls for Configuration	Defines the condition that must exist for this control to be used.

Text Value

Specify the script that is evaluated.

Remarks

If this element is used, you cannot specify the [PropertyName](#) element when defining the selection condition.

See Also

[PropertyName Element for ItemSelectionCondition for Controls for Configuration](#)

[ItemSelectionCondition Element for ExpressionBinding for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PropertyName Element for ExpressionBinding for Controls for Configuration

Article • 09/17/2021

Specifies the .NET property whose value is displayed by the common control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for Controls for Configuration	Defines the data that is displayed by the control.

Text Value

Specify the name of the .NET property whose value is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for ExpressionBinding for Controls for Configuration

Article • 09/17/2021

Specifies the script whose value is displayed by the common control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for Controls for Configuration	Defines the data that is displayed by the common control.

Text Value

Specify the script whose value is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Frame Element for CustomItem for Controls for Configuration

Article • 09/17/2021

Defines how the data is displayed, such as shifting the data to the left or right. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element

Syntax

XML

```
<Frame>
  <LeftIndent>NumberOfCharactersToShift</LeftIndent>
  <RightIndent>NumberOfCharactersToShift</RightIndent>
  <FirstLineHanging>NumberOfCharactersToShift</FirstLineHanging>
  <FirstLineIndent>NumberOfCharactersToShift</FirstLineIndent>
  <CustomItem>...</CustomItem>
</Frame>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Frame` element.

Attributes

None.

Child Elements

 Expand table

Element	Description
CustomItem Element	Required Element
FirstLineHanging Element for Frame for Controls for Configuration	<p>Optional element.</p> <p>Specifies how many characters the first line of data is shifted to the left.</p>
FirstLineIndent Element for Frame for Controls for Configuration	<p>Optional element.</p> <p>Specifies how many characters the first line of data is shifted to the right.</p>
LeftIndent Element for Frame for Controls for Configuration	<p>Optional element.</p> <p>Specifies how many characters the data is shifted away from the left margin.</p>
RightIndent Element for Frame for Controls for Configuration	<p>Optional element.</p> <p>Specifies how many characters the data is shifted away from the right margin.</p>

Parent Elements

 Expand table

Element	Description
CustomItem Element for CustomEntry for Controls for Configuration	Defines what data is displayed by the control and how it is displayed.

Remarks

You cannot specify the [FirstLineHanging](#) and the [FirstLineIndent](#) elements in the same [Frame](#) element.

See Also

[FirstLineHanging Element for Frame for Controls for Configuration](#)

FirstLineIndent Element for Frame for Controls for Configuration

LeftIndent Element for Frame for Controls for Configuration

RightIndent Element for Frame for Controls for Configuration

CustomItem Element for CustomEntry for Controls for Configuration

Writing a PowerShell Formatting File

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

FirstLineHanging Element for Frame for Controls for Configuration

Article • 09/17/2021

Specifies how many characters the first line of data is shifted to the left. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- FirstLineHanging Element

Syntax

XML

```
<FirstLineHanging>NumberOfCharactersToShift</FirstLineHanging>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `FirstLineHanging` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for Controls for Configuration	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the first line of the data.

Remarks

If this element is specified, you cannot specify the `FirstLineIndent` element.

See Also

[Frame Element for CustomItem for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

FirstLineIndent Element for Frame for Controls for Configuration

Article • 09/17/2021

Specifies how many characters the first line of data is shifted to the right. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- FirstLineIndent Element

Syntax

XML

```
<FirstLineIndent>NumberOfCharactersToShift</FirstLineIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `FirstLineIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for Controls for Configuration	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the first line of the data.

Remarks

If this element is specified, you cannot specify the [FirstLineHanging](#) element.

See Also

[FirstLineHanging Element for Frame for Controls for Configuration](#)

[Frame Element for CustomItem for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

LeftIndent Element for Frame for Controls for Configuration

Article • 09/17/2021

Specifies how many characters the data is shifted away from the left margin. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- LeftIndent Element

Syntax

XML

```
<LeftIndent>CharactersToShift</LeftIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `LeftIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for Controls for Configuration	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the data to the left.

Remarks

See Also

[Frame Element for CustomItem for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

RightIndent Element for Frame for Controls for Configuration

Article • 09/17/2021

Specifies how many characters the data is shifted away from the right margin. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- RightIndent Element

Syntax

XML

```
<RightIndent>CharactersToShift</RightIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `RightIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for Controls for Configuration	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the data to the right.

Remarks

See Also

[Frame Element for CustomItem for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

NewLine Element for CustomItem for Controls for Configuration

Article • 09/17/2021

Adds a blank line to the display of the control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- NewLine Element

Syntax

XML

```
<NewLine/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `NewLine` element.

Attributes

None.

Child Elements

None.

Parent Elements

[\[+\] Expand table](#)

Element	Description
CustomItem Element for CustomEntry for Controls for Configuration	Defines a control for the custom control view.

Remarks

See Also

[CustomItem Element for CustomEntry for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Text Element for CustomItem for Controls for Configuration

Article • 09/17/2021

Specifies text that is added to the data that is displayed by the control, such as a label, brackets to enclose the data, and spaces to indent the data. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Text Element

Syntax

XML

```
<Text>TextToDisplay</Text>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Text` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
CustomItem Element for CustomEntry for Controls for Configuration	Defines what data is displayed by the control and how it is displayed.

Text Value

Specify the text of a control for data that you want to display.

Remarks

See Also

[CustomItem Element for CustomEntry for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

EntrySelectedBy Element for CustomEntry for Controls

Article • 09/17/2021

Defines the .NET types that use the definition of the common control or the condition that must exist for this control to be used. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element

Syntax

XML

```
<EntrySelectedBy>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>SelectionSet</SelectionSetName>
  <SelectionCondition>...</SelectionCondition>
</EntrySelectedBy>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `EntrySelectedBy` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for Controls for Configuration	Optional element. Defines the condition that must exist for the common control definition to be used.
SelectionSetName Element for EntrySelectedBy for Controls for Configuration	Optional element. Specifies a set of .NET types that use this definition of the common control.
TypeName Element for EntrySelectedBy for Controls for Configuration	Optional element. Specifies a .NET type that uses this definition of the common control.

Parent Elements

[+] Expand table

Element	Description
CustomEntry Element for CustomControl for Controls for Configuration	Provides a definition of the common control.

Remarks

At a minimum, each definition must have at least one .NET type, selection set, or selection condition specified. There is no maximum limit to the number of types, selection sets, or selection conditions that you can specify.

See Also

[SelectionCondition Element for EntrySelectedBy for Controls for Configuration](#)

[SelectionSetName Element for EntrySelectedBy for Controls for Configuration](#)

[CustomEntry Element for CustomControl for Controls for Configuration](#)

[TypeName Element for EntrySelectedBy for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionCondition Element for EntrySelectedBy for Controls for Configuration

Article • 09/17/2021

Defines a condition that must exist for a common control definition to be used. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element

Syntax

XML

```
<SelectionCondition>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</SelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionCondition` element.

Attributes

None.

Child Elements

 Expand table

Element	Description
PropertyName Element for SelectionCondition for Controls for Configuration	Optional element. Specifies a .NET property that triggers the condition.
ScriptBlock Element for SelectionCondition for Controls for Configuration	Optional element. Specifies the script that triggers the condition.
SelectionSetName Element for SelectionCondition for Controls for Configuration	Optional element. Specifies the set of .NET types that triggers the condition.
TypeName Element for SelectionCondition for Controls for Configuration	Optional element. Specifies a .NET type that triggers the condition.

Parent Elements

 Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for Controls for Configuration	Defines the .NET types that use this entry of the common control definition.

Remarks

The following guidelines must be followed when defining a selection condition:

- The selection condition must specify at least one property name or a script block, but cannot specify both.
- The selection condition can specify any number of .NET types or selection sets, but cannot specify both.

For more information about how selection conditions can be used, see [Defining Conditions for when Data is Displayed](#).

See Also

[PropertyName Element for SelectionCondition for Controls for Configuration](#)

[ScriptBlock Element for SelectionCondition for Controls for Configuration](#)

[SelectionSetName Element for SelectionCondition for Controls for Configuration](#)

[TypeName Element for SelectionCondition for Controls for Configuration](#)

[EntrySelectedBy Element for CustomEntry for Controls for Configuration](#)

[Writing a Windows PowerShell Formatting and Types File](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



[PowerShell feedback](#)

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PropertyName Element for SelectionCondition for Controls for Configuration

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the entry is used. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for Controls for Configuration	Defines a condition that must exist for a common control definition to be used.

Text Value

Specify the .NET property name.

Remarks

The selection condition must specify at least one property name or a script, but cannot specify both. For more information about how selection conditions can be used, see [Defining Conditions for Displaying Data](#).

See Also

[SelectionCondition Element for EntrySelectedBy for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for SelectionCondition for Controls for Configuration

Article • 09/17/2021

Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the definition is used. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for Controls for Configuration	Defines a condition that must exist for the common control definition to be used.

Text Value

Specify the script that is evaluated.

Remarks

The selection condition must specify at least one script or property name to evaluate, but cannot specify both. For more information about how selection conditions can be used, see [Defining Conditions for Displaying Data](#).

See Also

[SelectionCondition Element for EntrySelectedBy for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for SelectionCondition for Controls for Configuration

Article • 09/17/2021

Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met, and the object is displayed by using this control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for Controls for Configuration	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the name of the selection set.

Remarks

Selection sets are common groups of .NET objects that can be used by any view that the formatting file defines. For more information about creating and referencing selection sets, see [Defining Sets of Objects](#).

The selection condition can specify a selection set or .NET type, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for Displaying Data](#).

See Also

[SelectionCondition Element for EntrySelectedBy for Controls for Configuration](#)

[Defining Conditions for When Data Is Displayed](#)

[Defining Selection Sets](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub



PowerShell feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for SelectionCondition for Controls for Configuration

Article • 09/17/2021

Specifies a .NET type that triggers the condition. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof.NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the [TypeName Element](#).

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for CustomEntry for Configuration	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

See Also

[SelectionCondition Element for EntrySelectedBy for CustomEntry for Configuration](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for EntrySelectedBy for Controls for Configuration

Article • 09/17/2021

Specifies a set of .NET types that use this definition of the control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element of Configuration
- Control Element for Controls for Configuration
- CustomControl Element for Control for Configuration
- CustomEntries Element for CustomControl for Configuration
- CustomEntry Element for CustomControl for Controls for Configuration
- EntrySelectedBy Element for CustomEntry for Controls for Configuration
- SelectionSetName Element for EntrySelectedBy for Controls for Configuration

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None

Child Elements

None.

Parent Elements

 Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for Controls for Configuration	Defines the .NET types that use this control definition or the condition that must exist for this definition to be used.

Text Value

Specify the name of the selection set.

Remarks

Each control definition must have at least one type name, selection set, or selection condition defined.

Selection sets are typically used when you want to define a group of objects that are used in multiple views. For more information about defining selection sets, see [Defining Selection Sets](#).

See Also

[EntrySelectedBy Element for CustomEntry for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for EntrySelectedBy for Controls for Configuration

Article • 09/17/2021

Specifies a .NET type that uses this definition of the control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof .NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for Controls for Configuration	Defines the .NET types that use this control definition or the condition that must exist for this definition to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

See Also

[EntrySelectedBy Element for CustomEntry for Controls for Configuration](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Name Element for Control for Controls for Configuration

Article • 09/17/2021

Specifies the name of the control. This element is used when defining a common control that can be used by all the views in the formatting file.

Schema

- Configuration Element
- Controls Element
- Control Element
- Name Element

Syntax

XML

```
<Name>NameOfControl</Name>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Name` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Control Element for Controls for Configuration	Defines a common control that can be used by all the views of the formatting file and the name that is used to reference the control.

Text Value

Specify the name that is used to reference this control.

Remarks

The name specified here can be used in the following elements to reference this control.

- When creating a table, list, wide or custom control view, the control can be specified by the following element: [GroupBy Element for View](#)
- When creating another common control, this control can be specified by the following element: [ExpressionBinding Element for CustomItem for Controls for Configuration](#)
- When creating a control that can be used by a view, this control can be specified by the following element: [ExpressionBinding Element for CustomItem for Controls for View](#)

See Also

[Control Element for Controls for Configuration](#)

[ExpressionBinding Element for CustomItem for Controls for Configuration](#)

[ExpressionBinding Element for CustomItem for Controls for View](#)

[GroupBy Element for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

DefaultSettings Element

Article • 09/17/2021

Defines common settings that apply to all the views of the formatting file. Common settings include displaying errors, wrapping text in tables, defining how collections are expanded, and more.

Schema

- Configuration Element
- DefaultSettings Element

Syntax

XML

```
<DefaultSettings>
  <ShowError/>
  <DisplayError/>
  <PropertyCountForTable>NumberOfProperties</PropertyCountForTable>
  <WrapTables/>
  <EnumerableExpansions>...</EnumerableExpansions>
</DefaultSettings>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `DefaultSettings` element.

Attributes

None.

Child Elements

 Expand table

Element	Description
DisplayError Element	<p>Optional element.</p> <p>Specifies that the string #ERR is displayed when an error occurs while displaying a piece of data.</p>
EnumerableExpansions Element	<p>Optional element.</p> <p>Defines the different ways that .NET objects are expanded when they are displayed in a view.</p>
PropertyCountForTable	<p>Optional element.</p> <p>Specifies the minimum number of properties that an object must have to display the object in a table view.</p>
ShowError Element	<p>Optional element.</p> <p>Specifies that the full error record is displayed when an error occurs while displaying a piece of data.</p>
WrapTables Element	<p>Optional element.</p> <p>Specifies that data in a table is moved to the next line if it does not fit into the width of the column.</p>

Parent Elements

[] [Expand table](#)

Element	Description
Configuration Element	Represents the top-level element of a formatting file.

Remarks

See Also

[Configuration Element](#)

[DisplayError Element](#)

[EnumerableExpansions Element](#)

[PropertyCountForTable](#)

[ShowError Element](#)

[WrapTables Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

DisplayError Element

Article • 09/17/2021

Specifies that the string #ERR is displayed when an error occurs displaying a piece of data.

Schema

- Configuration Element
- DefaultSettings Element
- DisplayError Element

Syntax

XML

```
<DisplayError/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `DisplayError` element.

Attributes

None.

Child Elements

None.

Parent Elements

Expand table

Element	Description
DefaultSettings	Defines common settings that apply to all the views of the formatting

Element	Description
Element	file.

Remarks

By default, when an error occurs while trying to display a piece of data, the location of the data is left blank. When this element is set to true, the #ERR string will be displayed.

See Also

[DefaultSettings Element](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

EnumerableExpansions Element

Article • 09/17/2021

Defines how .NET collection objects are expanded when they are displayed in a view.

Schema

- Configuration Element
- DefaultSettings Element
- EnumerableExpansions Element

Syntax

XML

```
<EnumerableExpansions>
  <EnumerableExpansion>...</EnumerableExpansion>
</EnumerableExpansions>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `EnumerableExpansions` element. There is no limit to the number of child elements that you can use.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
<code>EnumerableExpansion</code> Element	Optional element. Defines the specific .NET collection objects that are expanded when they are displayed in a view.

Parent Elements

[] [Expand table](#)

Element	Description
DefaultSettings Element	Defines common settings that apply to all the views of the formatting file.

Remarks

This element is used to define how collection objects and the objects in the collection are displayed. In this case, a collection object refers to any object that supports the [System.Collections.ICollection](#) interface.

See Also

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

EnumerableExpansion Element

Article • 09/17/2021

Defines how specific .NET collection objects are expanded when they are displayed in a view.

Schema

- Configuration Element
- DefaultSettings Element
- EnumerableExpansions Element
- EnumerableExpansion Element

Syntax

XML

```
<EnumerableExpansion>
  <EntrySelectedBy>...</EntrySelectedBy>
  <Expand>EnumOnly, CoreOnly, Both</Expand>
</EnumerableExpansion>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `EnumerableExpansion` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
<code>EntrySelectedBy</code> Element for <code>EnumerableExpansion</code>	Optional element.

Element	Description
	Defines which .NET collection objects are expanded by this definition.
Expand Element	Specifies how the collection object is expanded for this definition.

Parent Elements

 [Expand table](#)

Element	Description
EnumerableExpansions Element	Defines the different ways that .NET collection objects are expanded when they are displayed in a view.

Remarks

This element is used to define how collection objects and the objects in the collection are displayed. In this case, a collection object refers to any object that supports the [System.Collections.ICollection](#) interface.

The default behavior is to display only the properties of the objects in the collection.

See Also

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

EntrySelectedBy Element for EnumerableExpansion

Article • 09/17/2021

Defines the .NET types that use this definition or the condition that must exist for this definition to be used.

Schema

- Configuration Element
- DefaultSettings Element
- EnumerableExpansions Element
- EnumerableExpansion Element
- EntrySelectedBy Element for EnumerableExpansion

Syntax

XML

```
<EntrySelectedBy>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <SelectionCondition>...</SelectionCondition>
</EntrySelectedBy>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `EntrySelectedBy` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for EnumerableExpansion	Optional element. Defines the condition that must exist to expand the collection objects of this definition.
SelectionSetName Element for EntrySelectedBy for EnumerableExpansion	Optional element. Specifies a set of .NET types that use this definition of how collection objects are expanded.
TypeName Element for EntrySelectedBy for EnumerableExpansion	Optional element. Specifies a .NET type that uses this definition of how collection objects are expanded.

Parent Elements

[\[\] Expand table](#)

Element	Description
EnumerableExpansion Element	Defines how specific .NET collection objects are expanded when they are displayed in a view.

Remarks

You must specify at least one type, selection set, or selection condition for a definition entry. There is no maximum limit to the number of child elements that you can use.

Selection conditions are used to define a condition that must exist for the definition to be used, such as when an object has a specific property or that a specific property value or script evaluates to `true`. For more information about selection conditions, see

[Defining Conditions for Displaying Data](#).

See Also

[Defining Conditions for Displaying Data](#)

[EnumerableExpansion Element](#)

[SelectionCondition Element for EntrySelectedBy for EnumerableExpansion](#)

[SelectionSetName Element for EntrySelectedBy for EnumerableExpansion](#)

[TypeName Element for EntrySelectedBy for EnumerableExpansion](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionCondition Element for EntrySelectedBy for EnumerableExpansion

Article • 12/18/2023

Defines the condition that must exist to expand the collection objects of this definition.

Schema

- Configuration Element
- DefaultSettings Element
- EnumerableExpansions Element
- EnumerableExpansion Element
- EntrySelectedBy Element
- SelectionCondition Element

Syntax

XML

```
<SelectionCondition>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</SelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionCondition` element. You must specify a single `PropertyName` or `ScriptBlock` element. The `SelectionSetName` and `TypeName` elements are optional. You can specify one of either element.

Attributes

None.

Child Elements

[Expand table](#)

Element	Description
PropertyName Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion	Optional element. Specifies the .NET property that triggers the condition.
ScriptBlock Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion	Optional element. Specifies the script that triggers the condition.
SelectionSetName Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion	Optional element. Specifies the set of .NET types that triggers the condition.
TypeName Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion	Optional element. Specifies a .NET type that triggers the condition.

Parent Elements

[Expand table](#)

Element	Description
EntrySelectedBy Element for EnumerableExpansion	Defines which .NET collection objects are expanded by this definition.

Remarks

Each definition must have at least one type name, selection set, or selection condition defined.

When you are defining a selection condition, the following requirements apply:

- The selection condition must specify a least one property name or a script block, but cannot specify both.

- The selection condition can specify any number of .NET types or selection sets, but cannot specify both.

For more information about how to use selection conditions, see [Defining Conditions for Displaying Data](#).

For more information about other components of a wide view, see [Wide View](#).

See Also

[Defining Conditions for When Data Is Displayed](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PropertyName Element for SelectionCondition for Controls for View

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the entry is used. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element for Controls for View
- CustomControl Element for Control for Controls for View
- CustomEntries Element for CustomControl for Controls for View
- CustomEntry Element for CustomEntries for Controls for View
- EntrySelectedBy Element for CustomEntry for Controls for View
- SelectionCondition Element for EntrySelectedBy for Controls for View
- PropertyName Element for SelectionCondition for Controls for View

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for Controls for View	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the .NET property name.

Remarks

The selection condition must specify at least one property name or a script, but cannot specify both. For more information about how selection conditions can be used, see [Defining Conditions for Displaying Data](#).

See Also

[SelectionCondition Element for EntrySelectedBy for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for SelectionCondition for Controls for View

Article • 09/17/2021

Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the definition is used. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for Controls for View	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the script that is evaluated.

Remarks

The selection condition must specify at least one script or property name to evaluate, but cannot specify both. For more information about how selection conditions can be used, see [Defining Conditions for Displaying Data](#).

See Also

[SelectionCondition Element for EntrySelectedBy for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion

Article • 09/17/2021

Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met.

Schema

- Configuration Element
- DefaultSettings Element
- EnumerableExpansions Element
- EnumerableExpansions Element
- EntrySelectedBy Element
- SelectionCondition Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for EnumerableExpansion	Defines the condition that must exist to expand the collection objects of this definition.

Text Value

Specify the name of the selection set.

Remarks

The selection condition can specify a selection set or .NET type, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for Displaying Data](#).

Selection sets are common groups of .NET objects that can be used by any view that the formatting file defines. For more information about creating and referencing selection sets, see [Defining Selection Sets](#).

See Also

[Defining Selection Sets](#)

[SelectionCondition Element for EntrySelectedBy for EnumerableExpansion](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion

Article • 09/17/2021

Specifies a .NET type that triggers the condition.

Schema

- Configuration Element
- DefaultSettings Element
- EnumerableExpansions Element
- EnumerableExpansions Element
- EntrySelectedBy Element
- SelectionCondition Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof.NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] [Expand table](#)

Element	Description
SelectionCondition Element for EntrySelectedBy for EnumerableExpansion	Defines the condition that must exist to expand the collection objects of this definition.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO DirectoryInfo`.

Remarks

See Also

[SelectionCondition Element for EntrySelectedBy for EnumerableExpansion](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

SelectionSetName Element for EntrySelectedBy for EnumerableExpansion

Article • 09/17/2021

Specifies the set of .NET types that are expanded by this definition.

Schema

- Configuration Element
- DefaultSettings Element
- EnumerableExpansions Element
- EnumerableExpansion Element
- EntrySelectedBy Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
EntrySelectedBy Element for EnumerableExpansion	Defines the .NET collection objects that are expanded by this definition.

Text Value

Specify the name of the selection set.

Remarks

Each definition must specify one or more type names, a selection set, or a selection condition.

Selection sets are typically used when you want to define a group of objects that are used in multiple views. For example, you might want to create a table view and a list view for the same set of objects. For more information about defining selection sets, see [Defining Sets of Objects for a View](#).

See Also

[Defining Selection Sets](#)

[EntrySelectedBy Element for EnumerableExpansion](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for EntrySelectedBy for EnumerableExpansion

Article • 09/17/2021

Specifies a .NET type that is expanded by this definition. This element is used when defining a default settings.

Schema

- Configuration Element
- DefaultSettings Element
- EnumerableExpansions Element
- EnumerableExpansion Element
- EntrySelectedBy Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof .NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

Element	Description
EntrySelectedBy Element for EnumerableExpansion	Defines the .NET types that use this definition or the condition that must exist for this definition to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

See Also

[EntrySelectedBy Element for EnumerableExpansion](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Expand Element

Article • 09/17/2021

Specifies how the collection object is expanded for this definition.

Schema

- Configuration Element
- DefaultSettings Element
- EnumerableExpansions Element
- EnumerableExpansion Element
- Expand Element

Syntax

XML

```
<Expand>EnumOnly, CoreOnly, Both</Expand>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Expand` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] [Expand table](#)

Element	Description
EnumerableExpansion Element	Defines how specific .NET collection objects are expanded when they are displayed in a view.

Text Value

Specify one of the following values:

- `EnumOnly`: Displays only the properties of the objects in the collection.
- `CoreOnly`: Displays only the properties of the collection object.
- `Both`: Displays the properties of the objects in the collection and the properties of the collection object.

Remarks

This element is used to define how collection objects and the objects in the collection are displayed. In this case, a collection object refers to any object that supports the [System.Collections.ICollection](#) interface.

The default behavior is to display only the properties of the objects in the collection.

See Also

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

PropertyCountForTable Element

Article • 09/17/2021

Optional element. Specifies the minimum number of properties that an object must have to display the object in a table view.

Schema

- DefaultSettings Element
- PropertyCountForTable Element

Syntax

XML

```
<PropertyCountForTable>NumberOfProperties</PropertyCountForTable>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyCountForTable` element. The default value for this element is 4.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
DefaultSettings Element	Defines common settings that apply to all the views of the formatting file.

Remarks

See Also

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ShowError Element

Article • 09/17/2021

Specifies that the full error record is displayed when an error occurs while displaying a piece of data.

Schema

- Configuration Element
- DefaultSettings Element
- ShowError Element

Syntax

```
scr
<ShowError/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ShowError` element. The default value for this element is `false`.

Attributes

None.

Child Elements

None.

Parent Elements

Expand table

Element	Description
DefaultSettings	Defines common settings that apply to all the views of the formatting

Element	Description
Element	file.

Remarks

See Also

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

WrapTables Element

Article • 09/17/2021

Specifies that data in a table cell is moved to the next line if the data is longer than the width of the column.

Schema

- Configuration Element
- DefaultSettings Element
- WrapTables Element

Syntax

XML

```
<WrapTables/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `WrapTables` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
DefaultSettings	Defines common settings that apply to all the views of the formatting

Element	Description
Element	file.

Remarks

See Also

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSets Element

Article • 09/17/2021

Defines the common sets of .NET objects that can be used by all views of the formatting file. The views and controls of the formatting file can reference the complete set of objects by using only the name of the selection set.

Schema

- Configuration Element
- SelectionSets Element

Syntax

XML

```
<SelectionSets>
  <SelectionSet>...</SelectionSet>
</SelectionSets>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `SelectionSets` element. Each child element defines a set of objects that can be referenced by the name of the set. The order of the child elements is not significant.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
<code>SelectionSet</code> Element	Required element.

Element	Description
	Defines a single set of .NET objects that can be referenced by the name of the set.

Parent Elements

[\[+\] Expand table](#)

Element	Description
Configuration Element	Represents the top-level element of a formatting file.

Remarks

You can use selection sets when you have a set of related objects that you want to reference by using a single name, such as a set of objects that are related through inheritance. When defining your views, you can specify the set of objects by using the name of the selection set instead of listing all the objects within each view.

Common selection sets are specified by their name when defining the views of the formatting file or the definitions of the views. In these cases, the `SelectionSetName` child element of the `ViewSelectedBy` and `EntrySelectedBy` elements specifies the set to be used. For more information about selection sets, see [Defining Sets of Objects](#).

See Also

[Configuration Element](#)

[Defining Selection Sets](#)

[SelectionSet Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSet Element

Article • 09/17/2021

Defines a set of .NET objects that can be referenced by the name of the set.

Schema

- Configuration Element
- SelectionSets Element
- SelectionSet Element

Syntax

XML

```
<SelectionSet>
  <Name>SelectionSetName</Name>
  <Types>...</Types>
</SelectionSet>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `SelectionSet` element. Each selection set must have a name, and it must specify the .NET objects of the set.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
Name Element for SelectionSet	Required element. Specifies the name used to reference the selection set.

Element	Description
Types Element	<p>Required element.</p> <p>Defines the .NET objects that are in the selection set.</p>

Parent Elements

[\[+\] Expand table](#)

Element	Description
SelectionSets Element	Defines the common sets of .NET objects that can be used by all views
Format	of the formatting file.

Remarks

You can use selection sets when you have a set of related objects that you want to reference by using a single name, such as a set of objects that are related through inheritance. When defining your views, you can specify the set of objects by using the name of the selection set instead of listing all the objects within each view.

Common selection sets are specified by their name when defining the views of the formatting file or the definitions of the views. In these cases, the `SelectionSetName` child element of the `ViewSelectedBy` and `EntrySelectedBy` elements specifies the set to be used. For more information about selection sets, see [Defining Sets of Objects](#).

Example

The following example shows a `SelectionSet` element that defines four .NET types.

XML

```
<SelectionSets>
  <SelectionSet>
    <Name>FileSystemTypes</Name>
    <Types>
      <TypeName>System.IO.DirectoryInfo</TypeName>
      <TypeName>System.IO.FileInfo</TypeName>
      <TypeName>Deserialized.System.IO.DirectoryInfo</TypeName>
      <TypeName>Deserialized.System.IO.FileInfo</TypeName>
    </Types>
```

```
</SelectionSet>  
</SelectionSets>
```

See Also

[Defining Selection Sets](#)

[Name Element of SelectionSet](#)

[SelectionSets Element](#)

[Types Element](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Name Element for SelectionSet

Article • 09/17/2021

Specifies the name used to reference the selection set.

Schema

- Configuration Element
- SelectionSets Element
- SelectionSet Element
- Name Element

Syntax

XML

```
<Name>Name of selection set</Name>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `Name` Element.

Attributes

None.

Child Elements

None.

Parent Elements

Expand table

Element	Description
SelectionSet	Defines a single set of .NET objects that can be referenced by the name of

Element	Description
Element	the set.

Text Value

Specify the name to reference the selection set. There are no restrictions as to what characters can be used.

Remarks

The name specified here is used in the `SelectionSetName` element. The selection set that can be used by a view, by a definition of a view (views can have multiple definitions), or when specifying a selection condition. For more information about selection sets, see [Defining Sets of Objects](#).

Example

This example shows a `SelectionSet` element that defines four .NET types. The name of the selection set is "FileSystemTypes".

XML

```
<SelectionSets>
  <SelectionSet>
    <Name>FileSystemTypes</Name>
    <Types>
      <TypeName>System.IO.DirectoryInfo</TypeName>
      <TypeName>System.IO.FileInfo</TypeName>
      <TypeName>Deserialized.System.IO.DirectoryInfo</TypeName>
      <TypeName>Deserialized.System.IO.FileInfo</TypeName>
    </Types>
  </SelectionSet>
</SelectionSets>
```

See Also

[Defining Selection Sets](#)

[SelectionSet Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Types Element for SelectionSet

Article • 09/17/2021

Defines the .NET objects that are in the selection set.

Schema

- Configuration Element
- SelectionSets Element
- SelectionSet Element
- Types Element

Syntax

XML

```
<Types>
  <TypeName>Nameof.NetType</TypeName>
</Types>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `Types` element. There must be at least one child element, but there is no maximum limit to the number of child elements that can be added.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
TypeName Element of Types	Required element.

Element	Description
	Specifies the .NET object that belongs to the selection set.

Parent Elements

[+] Expand table

Element	Description
SelectionSet Element	Defines a set of .NET objects that can be referenced by the name of the set.

Remarks

The objects defined by this element make up a selection set that can be used by a view, by a definition of a view (views can have multiple definitions), or when specifying a selection condition. For more information about selection sets, see [Defining Sets of Objects](#).

Example

This example shows a `SelectionSet` element that defines four .NET types.

XML

```
<SelectionSets>
  <SelectionSet>
    <Name>FileSystemTypes</Name>
    <Types>
      <TypeName>System.IO.DirectoryInfo</TypeName>
      <TypeName>System.IO.FileInfo</TypeName>
      <TypeName>Deserialized.System.IO.DirectoryInfo</TypeName>
      <TypeName>Deserialized.System.IO.FileInfo</TypeName>
    </Types>
  </SelectionSet>
</SelectionSets>
```

See Also

[Defining Sets of Objects](#)

[SelectionSet Element](#)

TypeName Element of Types

Writing a PowerShell Formatting File

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for Types

Article • 09/17/2021

Specifies the .NET type of an object that belongs to the selection set.

Schema

- Configuration Element
- SelectionSets Element
- SelectionSet Element
- Types Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof.NetType</Name>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `TypeName` element. At least one `TypeName` element must be included in the selection set.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Types Element	Defines the .NET objects that are in the selection set.

Text Value

Specify the fully qualified name for the .NET type.

Remarks

You can use selection sets when you have a set of related objects that you want to reference by using a single name, such as a set of objects that are related through inheritance. When defining your views, you can specify the set of objects by using the name of the selection set instead of listing all the objects within each view.

Common selection sets are specified by their name when defining the views of the formatting file. In these cases, the `SelectionSetName` child element of the `ViewSelectedBy` element for the view specifies the set. However, different entries of a view can also specify a selection set that applies to only that entry of the view. For more information about selection sets, see [Defining Sets of Objects](#).

Example

The following example shows a `SelectionSet` element that defines four .NET types.

```
<SelectionSets>
  <SelectionSet>
    <Name>FileSystemTypes</Name>
    <Types>
      <TypeName>System.IO.DirectoryInfo</TypeName>
      <TypeName>System.IO.FileInfo</TypeName>
      <TypeName>Deserialized.System.IO.DirectoryInfo</TypeName>
      <TypeName>Deserialized.System.IO.FileInfo</TypeName>
    </Types>
  </SelectionSet>
</SelectionSets>
```

See Also

[Defining Selection Sets](#)

[SelectionSet Element](#)

[SelectionSets Element](#)

[Types Element](#)

[Writing a Windows PowerShell Formatting File](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ViewDefinitions Element

Article • 09/17/2021

Defines the views used to display .NET objects. These views can display the properties and script values of an object in a table format, list format, wide format, and custom control format.

Schema

- Configuration Element
- ViewDefinitions

Syntax

XML

```
<ViewDefinitions>
  <View>...</View>
</ViewDefinitions>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `ViewDefinitions` element. There is no limit to the number of views that can be defined in a formatting file, and they can be added in any order.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
View Element	Defines a view that is used to display one or more .NET objects.

Parent Elements

[] [Expand table](#)

Element	Description
Configuration Element	Represents the top-level element of a formatting file.

Remarks

For more information about the components of the different types of views, see the following topics:

- [Creating a Table View](#)
- [Creating a List View](#)
- [Creating a Wide View](#)
- [Custom Controls](#)

Example

This example shows a `ViewDefinitions` element that contains the parent elements for a table view and a list view.

XML

```
<Configuration>
  <ViewDefinitions>
    <View>
      <TableControl>...</TableControl>
    </View>
    <View>
      <ListControl>...</ListControl>
    </View>
  </ViewDefinitions>
</Configuration>
```

See Also

[Configuration Element](#)

[View Element](#)

[Creating a Table View](#)

[Creating a List View](#)

[Creating a Wide View](#)

[Custom Controls](#)

[Writing a PowerShell Formatting File](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

View Element

Article • 09/17/2021

Defines a view that displays one or more .NET objects. There is no limit to the number of views that can be defined in a formatting file.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element

Syntax

XML

```
<View>
  <Name>Friendly name of view.</Name>
  <OutOfBand />
  <ViewSelectedBy>...</ViewSelectedBy>
  <Controls>...</Controls>
  <GroupBy>...</GroupBy>
  <TableControl>...</TableControl>
  <ListControl>...</ListControl>
  <WideControl>...</WideControl>
  <CustomControl>...</CustomControl>
</View>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `View` element. You must specify one and only one of the control child elements, and you must specify the name of the view and the objects that use the view. Defining custom controls, how to group objects, and specifying if the view is out-of-band are optional.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
Controls Element for View	Optional element. Defines a set of controls that can be referenced by their name from within the view.
CustomControl Element	Optional element. Defines a custom control format for the view.
GroupBy Element for View	Optional element. Defines how the members of the .NET objects are grouped.
ListControl Element	Optional element. Defines a list format for the view.
Name Element for View	Required element. Specifies the name used to reference the view.
OutOfBand	Optional element When OutOfBand is true, the view applies regardless of previous objects that may have selected a different view.
TableControl Element	Optional element. Defines a table format for the view.
ViewSelectedBy Element for View	Required element. Defines the .NET objects that this view displays.
WideControl Element	Optional element. Defines a wide (single value) list format for the view.

Parent Elements

[+] Expand table

Element	Description
ViewDefinitions Element	Defines the views used to display objects.

Remarks

For more information about the components of different views and custom controls, see the following topics:

- [Table View Components](#)
- [List View Components](#)
- [Wide View Components](#)
- [Custom Controls](#)

Example

This example shows a `View` element that defines a table view for the [System.ServiceProcess.ServiceController](#) object.

XML

```
<ViewDefinitions>
  <View>
    <Name>service</Name>
    <ViewSelectedBy>
      <TypeName>System.ServiceProcess.ServiceController</TypeName>
    </ViewSelectedBy>
    <TableControl>...</TableControl>
  </View>
</ViewDefinitions>
```

See Also

[ViewDefinitions Element](#)

[Name Element for View](#)

[ViewSelectedBy Element](#)

[Controls Element for View](#)

[GroupBy Element for View](#)

[TableControl Element](#)

[ListControl Element](#)

[WideControl Element](#)

[CustomControl Element](#)

[Writing a PowerShell Formatting File](#)

Controls Element for View

Article • 09/17/2021

Defines the view controls that can be used by a specific view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element

Syntax

XML

```
<Controls>
  <Control>...</Control>
</Controls>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent elements of the `Controls` element. This element must have at least one child element. There is no maximum number of child elements, nor is their order significant.

Attributes

None.

Child Elements

[] Expand table

Element	Description
Control Element for Controls for View	Defines a control that can be used by the view.

Parent Elements

[+] Expand table

Element	Description
View Element	Defines a view that is used to display the members of one or more .NET objects.

Remarks

See Also

[Control Element](#)

[View Element](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Control Element for Controls for View

Article • 09/17/2021

Defines a control that can be used by the view and the name that is used to reference the control.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element

Syntax

XML

```
<Control>
  <Name>NameOfControl</Name>
  <CustomControl>...</CustomControl>
</Control>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `Control` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
Name Element for Control for View	Required element. Specifies the name of the control.
CustomControl Element for Control for Controls for View	Required element. Defines the control used by this view.

Parent Elements

[Expand table](#)

Element	Description
Controls Element	Defines the view controls that can be used by a specific view.

Remarks

This control can be specified by the following elements:

- [CustomControlName Element for ExpressionBinding for Controls for View](#)
- [CustomControlName Element for ExpressionBinding for CustomControl for View](#)
- [CustomControlName Element for ExpressionBinding for GroupBy](#)
- [CustomControlName Element for GroupBy](#)

See Also

[CustomControl Element for Control for Controls for View](#)

[CustomControlName Element for ExpressionBinding for Controls for View](#)

[CustomControlName Element for ExpressionBinding for CustomControl for View](#)

[CustomControlName Element for ExpressionBinding for GroupBy](#)

[CustomControlName Element for ExpressionBinding for GroupBy](#)

[Controls Element](#)

[Name Element for Control for Controls for View](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomControl Element for Control for Controls for View

Article • 09/17/2021

Defines a control that is used by the view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element

Syntax

XML

```
<CustomControl>
  <CustomEntries>...</CustomEntries>
</CustomControl>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomControl` element. You must specify only one child element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
CustomEntries Element for CustomControl for Controls for View	Required element. Provides the definitions for the control.

Parent Elements

[Expand table](#)

Element	Description
Control Element for Controls for View	Defines a control that can be used by the view and the name that is used to reference the control.

Remarks

See Also

[CustomEntries Element for CustomControl for View](#)

[Control Element for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

CustomEntries Element for CustomControl for Controls for View

Article • 09/17/2021

Provides the definitions for the control. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element

Syntax

XML

```
<CustomEntries>
  <CustomEntry>...</CustomEntry>
</CustomEntries>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent elements of the `CustomEntries` element. There is no maximum limit to the number of child elements that can be specified.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
CustomEntry Element for CustomEntries for Controls for View	Required element. Provides a definition of the control.

Parent Elements

[+] Expand table

Element	Description
CustomControl Element for Control for Controls for View	Defines the control used by the view.

Remarks

In most cases, a control has only one definition, which is specified in a single `CustomEntry` element. However, it is possible to provide multiple definitions if you want to use the same control to display different .NET objects. In those cases, you can define a `CustomEntry` element for each object or set of objects.

See Also

[CustomEntry Element for CustomEntries for Controls for View](#)

[CustomControl Element for Control for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomEntry Element for CustomEntries for Controls for View

Article • 09/17/2021

Provides a definition of the control. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element

Syntax

XML

```
<CustomEntry>
  <EntrySelectedBy>...</EntrySelectedBy>
  <CustomItem>...</CustomItem>
</CustomEntry>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent elements of the `CustomEntry` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for Controls for View	Optional element. Defines the .NET types that use this control definition or the condition that must exist for this definition to be used.
CustomItem Element for CustomEntry for Controls for View	Required element. Defines how the control displays the data.

Parent Elements

[+] Expand table

Element	Description
CustomEntries Element for CustomControl for View	Provides the definitions for the control.

Remarks

See Also

[CustomEntries Element for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

CustomItem Element for CustomEntry for Controls for View

Article • 09/17/2021

Defines what data is displayed by the control and how it is displayed. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element

Syntax

XML

```
<CustomItem>
  <ExpressionBinding>...</ExpressionBinding>
  <NewLine/>
  <Text>TextToDisplay</Text>
  <Frame>...<Frame>
</CustomItem>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomItem` element. For more information, see Remarks.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
ExpressionBinding Element for CustomItem for Controls for View	Optional element. Defines the data that is displayed by the control.
Frame Element for CustomItem for Controls for View	Optional element. Defines how the data is displayed, such as shifting the data to the left or right.
NewLine Element for CustomItem for Controls for View	Optional element. Adds a blank line to the display of the control.
Text Element for CustomItem for Controls for View	Optional element. Adds text, such as parentheses or brackets, to the display of the control.

Parent Elements

[+] Expand table

Element	Description
CustomEntry Element for CustomEntries for Controls for View	Provides a definition of the control.

Remarks

When specifying the child elements of the `CustomItem` element, keep the following in mind:

- The child elements must be added in the following sequence: `ExpressionBinding`, `NewLine`, `Text`, and `Frame`.
- There is no maximum limit to the number of sequences that you can specify.
- In each sequence, there is no maximum limit to the number of `ExpressionBinding` elements that you can use.

See Also

[ExpressionBinding Element for CustomItem for Controls for View](#)

[Frame Element for CustomItem for Controls for View](#)

[NewLine Element for CustomItem for Controls for View](#)

[Text Element for CustomItem for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ExpressionBinding Element for CustomItem for Controls for View

Article • 09/17/2021

Defines the data that is displayed by the control. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element

Syntax

XML

```
<ExpressionBinding>
  <CustomControl>...</CustomControl>
  <CustomControlName>NameofCommonCustomControl</CustomControlName>
  <EnumerateCollection/>
  <ItemSelectionCondition>...</ItemSelectionCondition>
  <PropertyName>Nameof .NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</ExpressionBinding>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ExpressionBinding` element.

Attributes

None.

Child Elements

[] [Expand table](#)

Element	Description
CustomControl Element	Optional element. Defines a control that is used by this control.
CustomControlName Element for ExpressionBinding for Controls for View	Optional element. Specifies the name of a common control or a view control.
EnumerateCollection Element for ExpressionBinding for Controls for View	Optional element. Specifies that the elements of collections are displayed.
ItemSelectionCondition Element of ExpressionBinding for Controls for View	Optional element. Defines the condition that must exist for this control to be used.
PropertyName Element for ExpressionBinding for Controls for View	Optional element. Specifies the .NET property whose value is displayed by the control.
ScriptBlock Element for ExpressionBinding for Controls for View	Optional element. Specifies the script whose value is displayed by the control.

Parent Elements

[] [Expand table](#)

Element	Description
CustomItem Element for CustomEntry for Controls for View	Defines what data is displayed by the control and how it is displayed.

Remarks

See Also

[CustomItem Element for CustomEntry for Controls for View](#)

[CustomControlName Element for ExpressionBinding for Controls for View](#)

[EnumerateCollection Element for ExpressionBinding for Controls for View](#)

[ItemSelectionCondition Element of ExpressionBinding for Controls for View](#)

[PropertyName Element for ExpressionBinding for Controls for View](#)

[ScriptBlock Element for ExpressionBinding for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomControlName Element for ExpressionBinding for Controls for View

Article • 09/17/2021

Specifies the name of a common control or a view control. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- CustomControlName Element

Syntax

XML

```
<CustomControlName>NameofCustomControl</CustomControlName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomControlName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for Controls for View	Defines the data that is displayed by the control.

Text Value

Specify the name of the control.

Remarks

You can create common controls that can be used by all the views of a formatting file, and you can create view controls that can be used by a specific view. The following elements specify the names of these controls:

- [Name Element for Control for Controls for Configuration](#)
- [Name Element for Control for Controls for View](#)

See Also

[Name Element for Control for Controls for Configuration](#)

[Name Element for Control for Controls for View](#)

[ExpressionBinding Element for CustomItem for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

EnumerateCollection Element for ExpressionBinding for Controls for View

Article • 09/17/2021

Specified that the elements of collections are displayed. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- EnumerateCollection Element

Syntax

XML

```
<EnumerateCollection/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `EnumerateCollection` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for Controls for View	Defines the data that is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ItemSelectionCondition Element for ExpressionBinding for Controls for View

Article • 09/17/2021

Defines the condition that must exist for this control to be used. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ItemSelectionCondition Element

Syntax

XML

```
<ItemSelectionCondition>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</ItemSelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ItemSelectionCondition` element.

Attributes

None.

Child Elements

[Expand table](#)

Element	Description
PropertyName Element for ItemSelectionCondition for Controls for View	Optional element. Specifies the .NET property that triggers the condition.
ScriptBlock Element for ItemSelectionCondition for Controls for View	Optional element. Specifies the script that triggers the condition.

Parent Elements

[Expand table](#)

Element	Description
ExpressionBinding Element for CustomItem for Controls for View	Defines the data that is displayed by the control.

Remarks

You can specify one property name or a script for this condition but cannot specify both.

See Also

[PropertyName Element for ItemSelectionCondition for Controls for View](#)

[ScriptBlock Element for ItemSelectionCondition for Controls for View](#)

[ExpressionBinding Element for CustomItem for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub



PowerShell feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PropertyName Element for ItemSelectionCondition for Controls for View

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the control is used. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ItemSelectionCondition Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ItemSelectionCondition Element of ExpressionBinding for Controls for View	Defines the condition that must exist for this control to be used.

Text Value

Specify the name of the .NET property that triggers the condition.

Remarks

If this element is used, you cannot specify the [ScriptBlock](#) element when defining the selection condition.

See Also

[ScriptBlock Element for ItemSelectionCondition for Controls for View](#)

[ItemSelectionCondition Element of ExpressionBinding for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

ScriptBlock Element for ItemSelectionCondition for Controls for View

Article • 09/17/2021

Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the control is used. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ItemSelectionCondition Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ItemSelectionCondition Element of ExpressionBinding for Controls for View	Defines the condition that must exist for this control to be used.

Text Value

Specify the script that is evaluated.

Remarks

If this element is used, you cannot specify the [PropertyName](#) element when defining the selection condition.

See Also

[PropertyName Element for ItemSelectionCondition for Controls for View](#)

[ItemSelectionCondition Element of ExpressionBinding for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

PropertyName Element for ExpressionBinding for Controls for View

Article • 09/17/2021

Specifies the .NET property whose value is displayed by the control. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for Controls for View	Defines the data that is displayed by the control.

Text Value

Specify the name of the .NET property whose value is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for ExpressionBinding for Controls for View

Article • 09/17/2021

Specifies the script whose value is displayed by the control. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for Controls for View	Defines the data that is displayed by the control.

Text Value

Specify the script whose value is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Frame Element for CustomItem for Controls for View

Article • 09/17/2021

Defines how the data is displayed, such as shifting the data to the left or right. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element

Syntax

XML

```
<Frame>
  <LeftIndent>NumberOfCharactersToShift</LeftIndent>
  <RightIndent>NumberOfCharactersToShift</RightIndent>
  <FirstLineHanging>NumberOfCharactersToShift</FirstLineHanging>
  <FirstLineIndent>NumberOfCharactersToShift</FirstLineIndent>
  <CustomItem>...</CustomItem>
</Frame>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Frame` element.

Attributes

None.

Child Elements

[+] [Expand table](#)

Element	Description
CustomItem Element	Required Element
FirstLineHanging Element of Frame of Controls of View	Optional element. Specifies how many characters the first line is shifted to the left.
FirstLineIndent Element of Frame of Controls of View	Optional element. Specifies how many characters the first line is shifted to the right.
LeftIndent Element of Frame of Controls of View	Optional element. Specifies how many characters the data is shifted away from the left margin.
RightIndent Element of Frame of Controls of View	Optional element. Specifies how many characters the data is shifted away from the right margin.

Parent Elements

[+] [Expand table](#)

Element	Description
CustomItem Element for CustomEntry for Controls for View	Defines what data is displayed by the control and how it is displayed.

Remarks

You cannot specify the [FirstLineHanging](#) and the [FirstLineIndent](#) elements in the same [Frame](#) element.

See Also

[FirstLineHanging Element of Frame of Controls of View](#)

[FirstLineIndent Element of Frame of Controls of View](#)

[LeftIndent Element of Frame of Controls of View](#)

[RightIndent Element of Frame of Controls of View](#)

[CustomItem Element for CustomEntry for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

FirstLineHanging Element for Frame for Controls for View

Article • 09/17/2021

Specifies how many characters the first line of data is shifted to the left. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- FirstLineHanging Element

Syntax

XML

```
<FirstLineHanging>NumberOfCharactersToShift</FirstLineHanging>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `FirstLineHanging` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for Controls for View	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the first line of the data.

Remarks

If this element is specified, you cannot specify the [FirstLineIndent](#) element.

See Also

[FirstLineIndent Element for Frame for Controls for View](#)

[Frame Element for CustomItem for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

FirstLineIndent Element for Frame for Controls for View

Article • 09/17/2021

Specifies how many characters the first line of data is shifted to the right. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- FirstLineIndent Element

Syntax

XML

```
<FirstLineIndent>NumberOfCharactersToShift</FirstLineIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `FirstLineIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for Controls for View	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the first line of the data.

Remarks

If this element is specified, you cannot specify the [FirstLineHanging](#) element.

See Also

[FirstLineHanging Element for Frame for Controls for View](#)

[Frame Element for CustomItem for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

LeftIndent Element for Frame for Controls for View

Article • 09/17/2021

Specifies how many characters the data is shifted away from the left margin. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- LeftIndent Element

Syntax

XML

```
<LeftIndent>CharactersToShift</LeftIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `LeftIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for Controls for View	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the data to the left.

Remarks

See Also

[Frame Element for CustomItem for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

RightIndent Element for Frame for Controls for View

Article • 09/17/2021

Specifies how many characters the data is shifted away from the right margin. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- RightIndent Element

Syntax

XML

```
<RightIndent>CharactersToShift</RightIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `RightIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for Controls for View	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the data to the right.

Remarks

See Also

[Frame Element for CustomItem for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

NewLine Element for CustomItem for Controls for View

Article • 09/17/2021

Adds a blank line to the display of the control. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- NewLine Element

Syntax

XML

```
<NewLine/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `NewLine` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
CustomItem Element for CustomEntry for Controls for View	Defines what data is displayed by the control and how it is displayed.

Remarks

See Also

[CustomItem Element for CustomEntry for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Text Element for CustomItem for Controls for View

Article • 09/17/2021

Specifies text that is added to the data that is displayed by the control, such as a label, brackets to enclose the data, and spaces to indent the data. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element

Syntax

XML

```
<Text>TextToDisplay</Text>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Text` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
CustomItem Element for CustomEntry for Controls for View	Defines what data is displayed by the control and how it is displayed.

Text Value

Specify the text of a control for data that you want to display.

Remarks

See Also

[CustomItem Element for CustomEntry for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

EntrySelectedBy Element for CustomEntry for Controls for View

Article • 09/17/2021

Defines the .NET types that use this control definition or the condition that must exist for this definition to be used. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element

Syntax

XML

```
<EntrySelectedBy>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <SelectionCondition>...</SelectionCondition>
</EntrySelectedBy>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `EntrySelectedBy` element. You must specify at least one type, selection set, or selection condition for a definition. There is no maximum limit to the number of child elements that you can use.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for Controls for View	Optional element. Defines the condition that must exist for this definition to be used.
SelectionSetName Element for EntrySelectedBy for Controls for View	Optional element. Specifies a set of .NET types that use this definition of the control.
TypeName Element for EntrySelectedBy for Controls for View	Optional element. Specifies a .NET type that uses this definition of the control.

Parent Elements

[+] Expand table

Element	Description
CustomEntry Element for CustomEntries for Controls for View	Provides a definition of the control.

Remarks

Selection conditions are used to define a condition that must exist for the definition to be used, such as when an object has a specific property or when a specific property value or script evaluates to `true`. For more information about selection conditions, see [Defining Conditions for when a View Entry or Item is Used](#).

See Also

[CustomEntry Element for CustomEntries for Controls for View](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionCondition Element for EntrySelectedBy for Controls for View

Article • 09/17/2021

Defines a condition that must exist for the control definition to be used. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element

Syntax

XML

```
<SelectionCondition>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</SelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionCondition` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
PropertyName Element for SelectionCondition for Controls for View	Optional element. Specifies a .NET property that triggers the condition.
ScriptBlock Element for SelectionCondition for Controls for View	Optional element. Specifies the script that triggers the condition.
SelectionSetName Element for SelectionCondition for Controls for View	Optional element. Specifies the set of .NET types that triggers the condition.
TypeName Element for SelectionCondition for Controls for View	Optional element. Specifies a .NET type that triggers the condition.

Parent Elements

[+] Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for Controls for View	Defines the .NET types that use this control definition or the condition that must exist for this definition to be used.

Remarks

When you are defining a selection condition, the following requirements apply:

- The selection condition must specify at least one property name or a script block, but cannot specify both.
- The selection condition can specify any number of .NET types or selection sets, but cannot specify both.

For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

See Also

[PropertyName Element for SelectionCondition for Controls for View](#)

[ScriptBlock Element for SelectionCondition for Controls for View](#)

[SelectionSetName Element for SelectionCondition for Controls for View](#)

[TypeName Element for SelectionCondition for Controls for View](#)

[EntrySelectedBy Element for CustomEntry for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PropertyName Element for SelectionCondition for CustomControl for View

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the definition is used. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- EntrySelectedBy Element
- SelectionCondition Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for CustomControl for View	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the .NET property name.

Remarks

The selection condition must specify at least one property name or a script, but cannot specify both. For more information about how selection conditions can be used, see [Defining Conditions for Displaying Data](#).

See Also

[SelectionCondition Element for EntrySelectedBy for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for SelectionCondition for CustomControl for View

Article • 09/17/2021

Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the definition is used. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- SelectionCondition Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for CustomControl for View	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the script that is evaluated.

Remarks

The selection condition must specify at least one script or property name to evaluate, but cannot specify both. For more information about how selection conditions can be used, see [Defining Conditions for Displaying Data](#).

See Also

[SelectionCondition Element for EntrySelectedBy for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for SelectionCondition for Controls for View

Article • 09/17/2021

Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met and the object is displayed using this control. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for Controls for View	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the name of the selection set.

Remarks

Selection sets are common groups of .NET objects that can be used by any view that the formatting file defines. For more information about creating and referencing selection sets, see [Defining Selection Sets](#).

The selection condition can specify a selection set or .NET type, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for Displaying Data](#).

See Also

[SelectionCondition Element for EntrySelectedBy for Controls for View](#)

[Defining Conditions for When Data Is Displayed](#)

[Defining Selection Sets](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub



PowerShell feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for SelectionCondition for Controls for View

Article • 09/17/2021

Specifies a .NET type that triggers the condition. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof.NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` Element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for Controls for View	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

See Also

[SelectionCondition Element for EntrySelectedBy for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for EntrySelectedBy for Controls for View

Article • 09/17/2021

Specifies a set of .NET types that use this definition of the control. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None

Child Elements

None.

Parent Elements

 Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for Controls for View	Defines the .NET types that use this control definition or the condition that must exist for this definition to be used.

Text Value

Specify the name of the selection set.

Remarks

Each control definition must have at least one type name, selection set, or selection condition defined.

Selection sets are typically used when you want to define a group of objects that are used in multiple views. For more information about defining selection sets, see [Defining Selection Sets](#).

See Also

[EntrySelectedBy Element for CustomEntry for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for EntrySelectedBy for Controls for View

Article • 09/17/2021

Specifies a .NET type that uses this definition of the control. This element is used when defining controls that can be used by a view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof .NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for Controls for View	Defines the .NET types that use this control definition or the condition that must exist for this definition to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

See Also

[EntrySelectedBy Element for CustomEntry for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Name Element for Control for Controls for View

Article • 09/17/2021

Specifies the name of the control.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Controls Element
- Control Element
- Name Element

Syntax

XML

```
<Name>ControlName</Name>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Name` element.

Attributes

None.

Child Elements

None.

Parent Elements

Element	Description
Control Element for Controls for View	Defines a control that can be used by the view and the name that is used to reference the control.

Text Value

Specify the name that is used to reference the control.

Remarks

The name specified here can be used in the following elements to reference this control.

- When creating a table, list, wide or custom control view, the control can be specified by the following element: [GroupBy Element for View](#)
- When creating another control that can be used by a view, this control can be specified by the following element: [ExpressionBinding Element for CustomItem for Controls for View](#)

See Also

[GroupBy Element for View](#)

[ExpressionBinding Element for CustomItem for Controls for View](#)

[Control Element for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomControl Element for View

Article • 09/17/2021

Defines a custom control format for the view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element

Syntax

XML

```
<CustomControl>
  <CustomEntries>...</CustomEntries>
</CustomControl>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomControl` element. You must specify one child element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
CustomEntries Element for CustomControl for View	Required element. Provides the definitions of the custom control view.

Parent Elements

[] [Expand table](#)

Element	Description
View Element	Defines a view that is used to display one or more .NET objects.

Remarks

In most cases, only one definition is required for each control view, but it is possible to provide multiple definitions if you want to use the same view to display different .NET objects. In those cases, you can provide a separate definition for each object or set of objects.

See Also

[CustomEntries Element for CustomControl for View](#)

[View Element](#)

[Writing a PowerShell Formatting File](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



[PowerShell feedback](#)

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomEntries Element for CustomControl for View

Article • 09/17/2021

Provides the definitions of the custom control view. The custom control view must specify one or more definitions.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element

Syntax

XML

```
<CustomEntries>
  <CustomEntry>...</CustomEntry>
</CustomEntries>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomControlEntries` element. You must specify one or more child elements.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
CustomEntry Element for CustomEntries for View	<p>Required element.</p> <p>Provides a definition of the custom control view.</p>

Parent Elements

[\[+\] Expand table](#)

Element	Description
CustomControl Element for View	<p>Required element.</p> <p>Defines a custom control format for the view.</p>

Remarks

In most cases, a control has only one definition, which is defined in a single `CustomEntry` element. However it is possible to have multiple definitions if you want to use the same control to display different .NET objects. In those cases, you can define a `CustomEntry` element for each object or set of objects.

See Also

[CustomControl Element for View](#)

[CustomEntry Element for CustomEntries for View](#)

[Writing a PowerShell Formatting File](#)



[Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



[PowerShell feedback](#)

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

more information, see [our contributor guide](#).

CustomEntry Element for CustomEntries for CustomControl for View

Article • 09/17/2021

Provides a definition of the custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element

Syntax

XML

```
<CustomEntry>
  <EntrySelectedBy>...</EntrySelectedBy>
  <CustomItem>...</CustomItem>
</CustomEntry>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomEntry` element. You must specify the items displayed by the definition.

Attributes

None.

Child Elements

[] Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for View	<p>Optional element.</p> <p>Defines the .NET types that use the definition of the custom control view or the condition that must exist for this definition to be used.</p>
CustomItem Element for CustomEntry for View	<p>Defines a control for the custom control definition.</p>

Parent Elements

[\[+\] Expand table](#)

Element	Description
CustomEntries Element for CustomControl for View	<p>Provides the definitions of the custom control view. The custom control view must specify one or more definitions.</p>

Remarks

In most cases, only one definition is required for each custom control view, but it is possible to have multiple definitions if you want to use the same view to display different .NET objects. In those cases, you can provide a separate definition for each object or set of objects.

See Also

[CustomControl Element for View](#)

[CustomItem Element for CustomEntry for View](#)

[EntrySelectedBy Element for CustomEntry for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

CustomItem Element for CustomEntry for CustomControl for View

Article • 09/17/2021

Defines what data is displayed by the custom control view and how it is displayed. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element

Syntax

XML

```
<CustomItem>
  <ExpressionBinding>...</ExpressionBinding>
  <Frame>...</Frame>
  <NewLine/>
  <Text>TextToDisplay</Text>
</CustomItem>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomItem` element.

Attributes

None.

Child Elements

[] Expand table

Element	Description
ExpressionBinding Element for CustomItem for CustomControl for View	Optional element. Defines the data that is displayed by the control.
Frame Element for CustomItem for CustomControl for View	Optional element. Defines what data is displayed by the custom control view and how it is displayed.
NewLine Element for CustomItem for Custom Control for View	Optional element. Adds a blank line to the display of the control.
Text Element for CustomItem for CustomControl for View	Optional element. Specifies additional text to the data displayed by the control.

Parent Elements

[] Expand table

Element	Description
CustomEntry Element for CustomEntries for CustomControl for View	Provides a definition of the custom control view.

Remarks

See Also

[CustomEntry Element for CustomEntries for View](#)

[ExpressionBinding Element for CustomItem for CustomControl for View](#)

[Frame Element for CustomItem for CustomControl for View](#)

[NewLine Element for CustomItem for CustomControl for View](#)

[Text Element for CustomItem for CustomControl for View](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ExpressionBinding Element for CustomItem for CustomControl for View

Article • 02/06/2023

Defines the data that is displayed by the control. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element

Syntax

XML

```
<ExpressionBinding>
  <CustomControl>...</CustomControl>
  <CustomControlName>NameofCommonCustomControl</CustomControlName>
  <EnumerateCollection/>
  <ItemSelectionCondition>...</ItemSelectionCondition>
  <PropertyName>Nameof.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</ExpressionBinding>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ExpressionBinding` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
CustomControl Element	Optional element. Defines a control that is used by this control.
CustomControlName Element for ExpressionBinding for CustomControl for View	Optional element. Specifies the name of a common control or a view control.
EnumerateCollection Element for ExpressionBinding for CustomControl for View	Optional element. Specified that the elements of collections are displayed.
ItemSelectionCondition Element for ExpressionBinding for CustomControl for View	Optional element. Defines the condition that must exist for this control to be used.
PropertyName Element for ExpressionBinding for CustomControl for View	Optional element. Specifies the .NET property whose value is displayed by the control.
ScriptBlock Element for ExpressionBinding for CustomCustomControl for View	Optional element. Specifies the script whose value is displayed by the control.

Parent Elements

[+] Expand table

Element	Description
CustomItem Element for CustomEntry for CustomControl for View	Defines what data is displayed by the custom control view and how it is displayed.

Remarks

See Also

[CustomControlName Element for ExpressionBinding for CustomControl for View](#)

[EnumerateCollection Element for ExpressionBinding for CustomControl for View](#)

[ItemSelectionCondition Element for ExpressionBinding for CustomControl for View](#)

[PropertyName Element for ExpressionBinding for CustomControl for View](#)

[ScriptBlock Element for ExpressionBinding for CustomControl for View](#)

[CustomItem Element for CustomEntry for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomControlName Element for ExpressionBinding for CustomControl for View

Article • 09/17/2021

Specifies the name of a common control or a view control. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- CustomControlName Element

Syntax

XML

```
<CustomControlName>NameofCustomControl</CustomControlName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomControlName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem	Defines the data that is displayed by the control.

Text Value

Specify the name of the control.

Remarks

You can create common controls that can be used by all the views of a formatting file and you can create view controls that can be used by a specific view. The names of these controls are specified by the following elements.

- [Name Element for Control for Controls for Configuration](#)
- [Name Element for Control for Controls for View](#)

See Also

[Name Element for Control for Controls for Configuration](#)

[Name Element for Control for Controls for View](#)

[ExpressionBinding Element for CustomItem](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

EnumerateCollection Element for ExpressionBinding for CustomControl for View

Article • 09/17/2021

Specifies that the elements of collections are displayed. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- EnumerateCollection Element

Syntax

XML

```
<EnumerateCollection/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `EnumerateCollection` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem	Defines the data that is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ItemSelectionCondition Element for ExpressionBinding for CustomControl

Article • 09/17/2021

Defines the condition that must exist for this control to be used. There is no limit to the number of selection conditions that can be specified for a control item. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ItemSelectionCondition Element

Syntax

XML

```
<ItemSelectionCondition>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</ItemSelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ItemSelectionCondition` element.

Attributes

None.

Child Elements

 Expand table

Element	Description
PropertyName Element for ItemSelectionCondition for CustomControl for View (Format)	Optional element. Specifies the .NET property that triggers the condition.
ScriptBlock Element for ItemSelectionCondition for CustomControl for View	Optional element. Specifies the script that triggers the condition.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for CustomControl for View	Defines the data that is displayed by the control.

Remarks

You can specify one property name or a script for this condition but cannot specify both.

See Also

[Writing a PowerShell Formatting File](#)

[ExpressionBinding Element for CustomItem for CustomControl for View](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

more information, see [our contributor guide](#).



[Provide product feedback](#)

PropertyName Element for ItemSelectionCondition for CustomControl for View

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the control is used. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element for CustomControl for View
- CustomEntry Element for CustomEntries for View
- CustomItem Element for CustomEntry for View
- ExpressionBinding Element for CustomItem for CustomControl for View
- ItemSelectionCondition Element for Expression Binding for CustomControl for View
- PropertyName Element for ItemSelectionCondition for CustomControl for View
(Format)

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] Expand table

Element	Description
ItemSelectionCondition Element for Expression Binding for CustomControl for View	Defines the condition that must exist for this control to be used.

Text Value

Specify the name of the .NET property that triggers the condition.

Remarks

If this element is used, you cannot specify the [ScriptBlock](#) element when defining the selection condition.

See Also

[ScriptBlock Element for ItemSelectionCondition for CustomControl for View](#)

[ItemSelectionCondition Element for Expression Binding for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

ScriptBlock Element for ItemSelectionCondition for CustomControl for View

Article • 09/17/2021

Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the control is used. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ItemSelectionCondition Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ItemSelectionCondition Element for Expression Binding for CustomControl for View	Defines the condition that must exist for this control to be used.

Text Value

Specify the script that is evaluated.

Remarks

If this element is used, you cannot specify the [PropertyName](#) element when defining the selection condition.

See Also

[PropertyName Element for ItemSelectionCondition for CustomControl for View](#)

[ItemSelectionCondition Element for Expression Binding for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PropertyName Element for ExpressionBinding for CustomControl for View

Article • 09/17/2021

Specifies the .NET property whose value is displayed by the control. This element is used when defining a custom control view

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for CustomControl for View	Defines the data that is displayed by the control.

Text Value

Specify the name of the .NET property whose value is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for ExpressionBinding for CustomControl for View

Article • 09/17/2021

Specifies the script whose value is displayed by the control. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for CustomControl for View	Defines the data that is displayed by the control.

Text Value

Specify the script whose value is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Frame Element for CustomItem for CustomControl for View

Article • 09/17/2021

Defines how the data is displayed, such as shifting the data to the left or right. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element

Syntax

XML

```
<Frame>
  <LeftIndent>NumberOfCharactersToShift</LeftIndent>
  <RightIndent>NumberOfCharactersToShift</RightIndent>
  <FirstLineHanging>NumberOfCharactersToShift</FirstLineHanging>
  <FirstLineIndent>NumberOfCharactersToShift</FirstLineIndent>
  <CustomItem>...</CustomItem>
</Frame>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Frame` element.

Attributes

None.

Child Elements

 Expand table

Element	Description
CustomItem Element	Required Element
FirstLineHanging Element	<p>Optional element.</p> <p>Specifies how many characters the first line of data is shifted to the left.</p>
FirstLineIndent Element	<p>Optional element.</p> <p>Specifies how many characters the first line of data is shifted to the right.</p>
LeftIndent Element	<p>Optional element.</p> <p>Specifies how many characters the data is shifted away from the left margin.</p>
RightIndent Element	<p>Optional element.</p> <p>Specifies how many characters the data is shifted away from the right margin.</p>

Parent Elements

 Expand table

Element	Description
CustomItem Element for CustomEntry for View	Defines what data is displayed by the control and how it is displayed.

Remarks

You cannot specify the [FirstLineHanging](#) and the [FirstLineIndent](#) elements in the same [Frame](#) element.

See Also

[FirstLineHanging Element](#)

FirstLineIndent Element

LeftIndent Element

RightIndent Element

CustomItem Element for CustomEntry for View

Writing a PowerShell Formatting File

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

FirstLineHanging Element for Frame for CustomControl

Article • 09/17/2021

Specifies how many characters the first line of data is shifted to the left. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- FirstLineHanging Element

Syntax

XML

```
<FirstLineHanging>NumberOfCharactersToShift</FirstLineHanging>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `FirstLineHanging` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for CustomControl for View	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the first line of the data.

Remarks

If this element is specified, you cannot specify the [FirstLineIndent](#) element.

See Also

[FirstLineIndent Element for Frame for CustomControl for View](#)

[Frame Element for CustomItem for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

FirstLineIndent Element for Frame for CustomControl for View

Article • 09/17/2021

Specifies how many characters the first line of data is shifted to the right. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- FirstLineIndent Element

Syntax

XML

```
<FirstLineIndent>NumberOfCharactersToShift</FirstLineIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `FirstLineIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] [Expand table](#)

Element	Description
Frame Element for CustomItem for CustomControl for View	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the first line of the data.

Remarks

If this element is specified, you cannot specify the [FirstLineHanging](#) element.

See Also

[FirstLineHanging Element for Frame for CustomControl for View](#)

[Frame Element for CustomItem for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



[PowerShell feedback](#)

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

LeftIndent Element for Frame for CustomControl for View

Article • 09/17/2021

Specifies how many characters the data is shifted away from the left margin. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- LeftIndent Element

Syntax

XML

```
<LeftIndent>CharactersToShift</LeftIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `LeftIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for CustomControl for View	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the data to the left.

Remarks

See Also

[Frame Element for CustomItem for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

RightIndent Element for Frame for CustomControl for View

Article • 09/17/2021

Specifies how many characters the data is shifted away from the right margin. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- RightIndent Element

Syntax

XML

```
<RightIndent>CharactersToShift</RightIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `RightIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for CustomControl for View	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the data to the right.

Remarks

See Also

[Frame Element for CustomItem for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

NewLine Element for CustomItem for CustomControl for View

Article • 09/17/2021

Adds a blank line to the display of the control. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- NewLine Element

Syntax

XML

```
<NewLine/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `NewLine` element.

Attributes

None.

Child Elements

None.

Parent Elements

[Expand table](#)

Element	Description
CustomItem Element for CustomEntry for View	Defines a control for the custom control view.

Remarks

See Also

[CustomItem Element for CustomEntry for View](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Text Element for CustomItem for CustomView for View

Article • 09/17/2021

Specifies text that is added to the data that is displayed by the control, such as a label, brackets to enclose the data, and spaces to indent the data. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Text Element

Syntax

XML

```
<Text>TextToDisplay</Text>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Text` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
CustomItem Element for CustomEntry for View	Defines a control for the custom control view.

Text Value

Specify the text of a control for data that you want to display.

Remarks

See Also

[CustomItem Element for CustomEntry for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

EntrySelectedBy Element for CustomEntry for CustomControl for View

Article • 09/17/2021

Defines the .NET types that use this custom entry or the condition that must exist for this entry to be used.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element

Syntax

XML

```
<EntrySelectedBy>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <SelectionCondition>...</SelectionCondition>
</EntrySelectedBy>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `EntrySelectedBy` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for CustomEntry	Optional element. Defines the condition that must exist for this definition to be used.
SelectionSetName Element for EntrySelectedBy for CustomEntry	Optional element. Specifies a set of .NET types that use this definition of the control view.
TypeName Element for EntrySelectedBy for CustomEntry	Optional element. Specifies a .NET type that uses this definition of the control view.

Parent Elements

[+] Expand table

Element	Description
CustomEntry Element for CustomEntries for View	Defines the controls used by specific .NET objects.

Remarks

You must specify at least one type, selection set, or selection condition for an entry. There is no maximum limit to the number of child elements that you can use.

Selection conditions are used to define a condition that must exist for the entry to be used, such as when an object has a specific property or when a specific property value or script evaluates to `true`. For more information about selection conditions, see [Defining Conditions for when a View Entry or Item is Used](#).

For more information about the components of a custom control view, see [Custom Control View](#).

See Also

[SelectionCondition Element for EntrySelectedBy for CustomEntry](#)

[SelectionSetName Element for EntrySelectedBy for CustomEntry](#)

[TypeName Element for EntrySelectedBy for CustomEntry](#)

[CustomEntry Element for CustomEntries for View](#)

[Custom Control View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

SelectionCondition Element for EntrySelectedBy for CustomControl

Article • 02/06/2023

Defines a condition that must exist for a control definition to be used. This element is used when defining a custom control view.

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- EntrySelectedBy Element
- SelectionCondition Element

Syntax

XML

```
<SelectionCondition>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</SelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionCondition` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
PropertyName Element for SelectionCondition for CustomControl for View	Optional element. Specifies a .NET property that triggers the condition.
ScriptBlock Element for SelectionCondition for CustomControl for View	Optional element. Specifies the script that triggers the condition.
SelectionSetName Element for SelectionCondition for Custom Control for View	Optional element. Specifies the set of .NET types that triggers the condition.
TypeName Element for SelectionCondition for CustomControl for View	Optional element. Specifies a .NET type that triggers the condition.

Parent Elements

[+] Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for CustomControl for View	Defines the .NET types that use this control definition or the condition that must exist for this definition to be used.

Remarks

When you are defining a selection condition, the following requirements apply:

- The selection condition must specify at least one property name or a script block, but cannot specify both.
- The selection condition can specify any number of .NET types or selection sets, but cannot specify both.

For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

See Also

[PropertyName Element for SelectionCondition for CustomControl for View](#)

[ScriptBlock Element for SelectionCondition for CustomControl for View](#)

[SelectionSetName Element for SelectionCondition for Custom Control for View](#)

[TypeName Element for SelectionCondition for CustomControl for View](#)

[EntrySelectedBy Element for CustomEntry for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

PropertyName Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the definition is used.

Schema

- Configuration Element
- DefaultSettings Element
- EnumerableExpansions Element
- EnumerableExpansion Element
- EntrySelectedBy Element
- SelectionCondition Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for EnumerableExpansion	Defines the condition that must exist to expand the collection objects of this definition.

Text Value

Specify the .NET property name.

Remarks

The selection condition must specify at least one property name or a script to evaluate, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

See Also

[Defining Conditions for When Data is Displayed](#)

[ScriptBlock Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion](#)

[SelectionCondition Element for EntrySelectedBy for EnumerableExpansion](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion

Article • 09/17/2021

Specifies the script that triggers the condition.

Schema

- Configuration Element
- DefaultSettings Element
- EnumerableExpansions Element
- EnumerableExpansion Element
- EntrySelectedBy Element
- SelectionCondition Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] [Expand table](#)

Element	Description
SelectionCondition Element for EntrySelectedBy for EnumerableExpansion	Defines the condition that must exist to expand the collection objects of this definition.

Text Value

Specify the script that is evaluated.

Remarks

The selection condition must specify at least one script or property name to evaluate, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

See Also

[Defining Conditions for When Data Is Displayed](#)

[PropertyName Element for SelectionCondition for EntrySelectedBy for EnumerableExpansion](#)

[SelectionCondition Element for EntrySelectedBy for EnumerableExpansion](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for SelectionCondition for CustomControl for View

Article • 09/17/2021

Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met and the object is displayed using this control. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for CustomControl for View	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the name of the selection set.

Remarks

Selection sets are common groups of .NET objects that can be used by any view that the formatting file defines. For more information about creating and referencing selection sets, see [Defining Sets of Objects](#).

The selection condition can specify a selection set or .NET type, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for When Data is Displayed](#).

See Also

[SelectionCondition Element for EntrySelectedBy for CustomControl for View](#)

[Defining Conditions for When Data Is Displayed](#)

[Defining Selection Sets](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

TypeName Element for SelectionCondition for CustomControl for View

Article • 09/17/2021

Specifies a .NET type that triggers the condition. This element is used when defining a custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- SelectionCondition Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof.NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the [TypeName Element](#).

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for CustomControl for View	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

See Also

[SelectionCondition Element for EntrySelectedBy for CustomControl for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for EntrySelectedBy for CustomControl for View

Article • 09/17/2021

Specifies a set of .NET objects for the list entry. There is no limit to the number of selection sets that can be specified for an entry.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for View	Defines the .NET types that use this custom entry or the condition that must exist for this entry to be used.

Text Value

Specify the name of the selection set.

Remarks

Each custom control entry must have at least one type name, selection set, or selection condition defined.

Selection sets are typically used when you want to define a group of objects that are used in multiple views. For example, you might want to create a table view and a list view for the same set of objects. For more information about defining selection sets, see [Defining Selection Sets](#).

For more information about the components of a custom control view, see [Creating Custom Controls](#).

See Also

[EntrySelectedBy Element for CustomEntry for View](#)

[Custom Control View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub



PowerShell feedback

PowerShell is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for EntrySelectedBy for CustomEntry for View

Article • 09/17/2021

Specifies a .NET type that uses this definition of the custom control view. There is no limit to the number of types that can be specified for a definition.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof .NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for View	Defines the .NET types that use this custom control view definition or the condition that must exist for this definition to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

Each custom control view definition must have at least one type name, selection set, or selection condition defined.

For more information about the components of a custom control view, see [Creating Custom Controls](#).

See Also

[Creating Custom Controls](#)

[EntrySelectedBy Element for CustomEntry for View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

GroupBy Element for View

Article • 09/17/2021

Defines how a new group of objects is displayed. This element is used when defining a table, list, wide, or custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element

Syntax

XML

```
<GroupBy>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
  <Label>TextToDisplay</Label>
  <CustomControl>...</CustomControl>
  <CustomControlName>NameOfControl</CustomControlName>
</GroupBy>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent elements.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
CustomControl Element for GroupBy	Optional element. Defines the custom control that displays new groups.
CustomControlName Element for GroupBy	Optional element. Specifies the name of a control that is used to display the new group.
Label Element for GroupBy	Optional element. Specifies a label that is displayed when a new group is encountered.
PropertyName Element for GroupBy	Optional element. Specifies the .NET property that starts a new group whenever its value changes.
ScriptBlock Element for GroupBy	Optional element. Specifies the script that starts a new group whenever its value changes.

Parent Elements

[View Element](#) Expand table

Element	Description
View Element	Defines a view that displays one or more .NET objects.

Remarks

When defining how a new group of objects is displayed, you must specify the property or script that will start the new group; however, you cannot specify both.

See Also

[CustomControlName Element for GroupBy](#)

[Label Element for GroupBy](#)

[PropertyName Element for GroupBy](#)

[ScriptBlock Element for GroupBy](#)

[View Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomControl Element for GroupBy

Article • 09/17/2021

Defines the custom control that displays the new group.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element

Syntax

XML

```
<CustomControl>
  <CustomEntries>...</CustomEntries>
<CustomControl>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `CustomControl` element. You can specify any number of child elements and list them in any order.

Attributes

None.

Child Elements

[\[+\] Expand table](#)

Element	Description
CustomEntries Element for CustomControl for GroupBy	Required element.

Element	Description
	Provides the definitions for the control.

Parent Elements

[\[+\] Expand table](#)

Element	Description
GroupBy Element for View	Defines how Windows PowerShell displays a new group of objects.

Remarks

See Also

[CustomEntries Element for CustomControl for GroupBy](#)

[GroupBy Element for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

CustomEntries Element for CustomControl for GroupBy

Article • 09/17/2021

Provides the definitions for the control. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element

Syntax

XML

```
<CustomEntries>
  <CustomEntry>...</CustomEntry>
</CustomEntries>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent elements of the `CustomEntries` element. There is no maximum limit to the number of child elements that can be specified.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
CustomEntry Element for CustomControl for GroupBy	Required element. Provides a definition of the control.

Parent Elements

[\[+\] Expand table](#)

Element	Description
CustomControl Element for GroupBy	Defines the custom control that displays the new group.

Remarks

In most cases, a control has only one definition, which is specified in a single `CustomEntry` element. However, it is possible to provide multiple definitions if you want to use the same control to display different groups. In those cases, you can define a `CustomEntry` element for a group.

See Also

[CustomEntry Element for CustomEntries for Controls for View](#)

[CustomControl Element for GroupBy](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomEntry Element for CustomControl for GroupBy

Article • 09/17/2021

Provides a definition of the control. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element

Syntax

XML

```
<CustomEntry>
  <EntrySelectedBy>...</EntrySelectedBy>
  <CustomItem>...</CustomItem>
</CustomEntry>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent elements of the `CustomEntry` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for GroupBy	Optional element. Defines the .NET types that use this control definition or the condition that must exist for this definition to be used.
CustomItem Element for CustomEntry for GroupBy	Required element. Defines how the control displays the data.

Parent Elements

[+] Expand table

Element	Description
CustomEntries Element for CustomControl for GroupBy	Provides the definitions for the control.

Remarks

See Also

[EntrySelectedBy Element for CustomEntry for GroupBy](#)

[CustomItem Element for CustomEntry for GroupBy](#)

[CustomEntries Element for CustomControl for GroupBy](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomItem Element for CustomEntry for GroupBy

Article • 09/17/2021

Defines what data is displayed by the custom control view and how it is displayed. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomItem Element

Syntax

XML

```
<CustomItem>
  <ExpressionBinding>...</ExpressionBinding>
  <Frame>...</Frame>
  <NewLine/>
  <Text>TextToDisplay</Text>
</CustomItem>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomItem` element.

Attributes

None.

Child Elements

[] Expand table

Element	Description
ExpressionBinding Element for CustomItem for GroupBy	Optional element. Defines the data that is displayed by the control.
Frame Element for CustomItem for GroupBy	Optional element. Defines what data is displayed by the custom control view and how it is displayed.
NewLine Element for CustomItem for GroupBy	Optional element. Adds a blank line to the display of the control.
Text Element for CustomItem for GroupBy	Optional element. Specifies additional text to the data displayed by the control.

Parent Elements

[] Expand table

Element	Description
CustomEntry Element for CustomControl for GroupBy	Provides a definition of the custom control view.

Remarks

See Also

[CustomEntry Element for CustomControl for GroupBy](#)

[ExpressionBinding Element for CustomItem for GroupBy](#)

[Frame Element for CustomItem for GroupBy](#)

[NewLine Element for CustomItem for GroupBy](#)

[Text Element for CustomItem for GroupBy](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ExpressionBinding Element for CustomItem for GroupBy

Article • 09/17/2021

Defines the data that is displayed by the control. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element

Syntax

XML

```
<ExpressionBinding>
  <CustomControl>...</CustomControl>
  <CustomControlName>NameofCommonCustomControl</CustomControlName>
  <EnumerateCollection/>
  <ItemSelectionCondition>...</ItemSelectionCondition>
  <PropertyName>Nameof .NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</ExpressionBinding>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ExpressionBinding` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
CustomControl Element	Optional element. Defines a control that is used by this control.
CustomControlName Element for ExpressionBinding for GroupBy	Optional element. Specifies the name of a common control or a view control.
EnumerateCollection Element for ExpressionBinding for GroupBy EnumerateCollection Element for ExpressionBinding for GroupBy	Optional element. Specified that the elements of collections are displayed.
ItemSelectionCondition Element for ExpressionBinding for GroupBy	Optional element. Defines the condition that must exist for this control to be used.
PropertyName Element for ExpressionBinding for GroupBy	Optional element. Specifies the .NET property whose value is displayed by the control.
ScriptBlock Element for ExpressionBinding for GroupBy	Optional element. Specifies the script whose value is displayed by the control.

Parent Elements

[+] Expand table

Element	Description
CustomItem Element for CustomEntry for GroupBy	Defines what data is displayed by the custom control view and how it is displayed.

See Also

[CustomControlName Element for ExpressionBinding for GroupBy](#)

[EnumerateCollection Element for ExpressionBinding for GroupBy](#)

[ItemSelectionCondition Element for ExpressionBinding for GroupBy](#)

[PropertyName Element for ExpressionBinding for GroupBy](#)

[ScriptBlock Element for ExpressionBinding for GroupBy](#)

[CustomItem Element for CustomEntry for GroupBy](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

CustomControlName Element for ExpressionBinding for GroupBy

Article • 09/17/2021

Specifies the name of a common control or a view control. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- CustomControlName Element

Syntax

XML

```
<CustomControlName>NameofCustomControl</CustomControlName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `CustomControlName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for GroupBy	Defines the data that is displayed by the control.

Text Value

Specify the name of the control.

Remarks

You can create common controls that can be used by all the views of a formatting file, and you can create view controls that can be used by a specific view. The following elements specify the names of these controls:

- [Name Element for Control for Controls for Configuration](#)
- [Name Element for Control for Controls for View](#)

See Also

[Name Element for Control for Controls for Configuration](#)

[Name Element for Control for Controls for View](#)

[ExpressionBinding Element for CustomItem for GroupBy](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

EnumerateCollection Element for ExpressionBinding for GroupBy

Article • 09/17/2021

Specifies that the elements of collections are displayed. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- EnumerateCollection Element

Syntax

XML

```
<EnumerateCollection/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `EnumerateCollection` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for GroupBy	Defines the data that is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem for GroupBy](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ItemSelectionCondition Element for ExpressionBinding for GroupBy

Article • 09/17/2021

Defines the condition that must exist for this control to be used. There is no limit to the number of selection conditions that can be specified for a control item. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ItemSelectionCondition Element

Syntax

XML

```
<ItemSelectionCondition>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</ItemSelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ItemSelectionCondition` element.

Attributes

None.

Child Elements

[] Expand table

Element	Description
PropertyName Element for ItemSelectionCondition for GroupBy	Optional element. Specifies the .NET property that triggers the condition.
ScriptBlock Element for ItemSelectionCondition for GroupBy	Optional element. Specifies the script that triggers the condition.

Parent Elements

[] Expand table

Element	Description
ExpressionBinding Element for CustomItem for GroupBy	Defines the data that is displayed by the control.

Remarks

You can specify one property name or a script for this condition but cannot specify both.

See Also

[Writing a PowerShell Formatting File](#)

[ExpressionBinding Element for CustomItem for GroupBy](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

more information, see [our contributor guide](#).



[Provide product feedback](#)

PropertyName Element for ItemSelectionCondition for GroupBy

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the control is used. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ItemSelectionCondition Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
ItemSelectionCondition Element for ExpressionBinding for GroupBy	Defines the condition that must exist for this control to be used.

Text Value

Specify the name of the .NET property that triggers the condition.

Remarks

If this element is used, you cannot specify the [ScriptBlock](#) element when defining the selection condition.

See Also

[ScriptBlock Element for ItemSelectionCondition for GroupBy](#)

[ItemSelectionCondition Element for ExpressionBinding for GroupBy](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

ScriptBlock Element for ItemSelectionCondition for GroupBy

Article • 09/17/2021

Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the control is used. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ItemSelectionCondition Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ItemSelectionCondition Element for ExpressionBinding for GroupBy	Defines the condition that must exist for this control to be used.

Text Value

Specify the script that is evaluated.

Remarks

If this element is used, you cannot specify the [PropertyName](#) element when defining the selection condition.

See Also

[ItemSelectionCondition Element for ExpressionBinding for GroupBy](#)

[PropertyName Element for ItemSelectionCondition for GroupBy](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PropertyName Element for ExpressionBinding for GroupBy

Article • 09/17/2021

Specifies the .NET property whose value is displayed by the control. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for GroupBy	Defines the data that is displayed by the control.

Text Value

Specify the name of the .NET property whose value is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem for GroupBy](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for ExpressionBinding for GroupBy

Article • 09/17/2021

Specifies the script whose value is displayed by the control. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- ExpressionBinding Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ExpressionBinding Element for CustomItem for GroupBy	Defines the data that is displayed by the control.

Text Value

Specify the script whose value is displayed by the control.

Remarks

See Also

[ExpressionBinding Element for CustomItem for GroupBy](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Frame Element for CustomItem for GroupBy

Article • 09/17/2021

Defines how the data is displayed, such as shifting the data to the left or right. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element

Syntax

XML

```
<Frame>
  <LeftIndent>NumberOfCharactersToShift</LeftIndent>
  <RightIndent>NumberOfCharactersToShift</RightIndent>
  <FirstLineHanging>NumberOfCharactersToShift</FirstLineHanging>
  <FirstLineIndent>NumberOfCharactersToShift</FirstLineIndent>
  <CustomItem>...</CustomItem>
</Frame>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Frame` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
CustomItem Element	Required Element
FirstLineHanging Element for Frame for GroupBy	Optional element. Specifies how many characters the first line of data is shifted to the left.
FirstLineIndent Element for Frame for GroupBy	Optional element. Specifies how many characters the first line of data is shifted to the right.
LeftIndent Element for Frame for GroupBy	Optional element. Specifies how many characters the data is shifted away from the left margin.
RightIndent Element for Frame for GroupBy	Optional element. Specifies how many characters the data is shifted away from the right margin.

Parent Elements

[+] Expand table

Element	Description
CustomItem Element for CustomEntry for GroupBy	Defines what data is displayed by the control and how it is displayed.

Remarks

You cannot specify the [FirstLineHanging](#) and the [FirstLineIndent](#) elements in the same [Frame](#) element.

See Also

[FirstLineHanging Element for Frame for GroupBy](#)

[FirstLineIndent Element for Frame for GroupBy](#)

[LeftIndent Element for Frame for GroupBy](#)

[RightIndent Element for Frame for GroupBy](#)

[CustomItem Element for CustomEntry for GroupBy](#)

[Writing a PowerShell Formatting File](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

FirstLineHanging Element for Frame for GroupBy

Article • 09/17/2021

Specifies how many characters the first line of data is shifted to the left. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- FirstLineHanging Element

Syntax

XML

```
<FirstLineHanging>NumberOfCharactersToShift</FirstLineHanging>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `FirstLineHanging` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for GroupBy	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the first line of the data.

Remarks

If this element is specified, you cannot specify the [FirstLineIndent](#) element.

See Also

[FirstLineIndent Element for Frame for GroupBy](#)

[Frame Element for CustomItem for GroupBy](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

FirstLineIndent Element for Frame for GroupBy

Article • 09/17/2021

Specifies how many characters the first line of data is shifted to the right. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element for View
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- FirstLineIndent Element

Syntax

XML

```
<FirstLineIndent>NumberOfCharactersToShift</FirstLineIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `FirstLineIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for GroupBy	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the first line of the data.

Remarks

If this element is specified, you cannot specify the [FirstLineHanging](#) element.

See Also

[FirstLineHanging Element for Frame for GroupBy](#)

[Frame Element for CustomItem for GroupBy](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

LeftIndent Element for Frame

Article • 09/17/2021

Specifies how many characters the data is shifted away from the left margin. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- LeftIndent Element

Syntax

XML

```
<LeftIndent>CharactersToShift</LeftIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `LeftIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] [Expand table](#)

Element	Description
Frame Element for CustomItem for GroupBy	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the data to the left.

Remarks

See Also

[Frame Element for CustomItem for GroupBy](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

RightIndent Element for Frame for GroupBy

Article • 09/17/2021

Specifies how many characters the data is shifted away from the right margin. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Frame Element
- RightIndent Element

Syntax

XML

```
<RightIndent>CharactersToShift</RightIndent>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `RightIndent` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
Frame Element for CustomItem for GroupBy	Defines how the data is displayed, such as shifting the data to the left or right.

Text Value

Specify the number of characters that you want to shift the data to the right.

Remarks

See Also

[Frame Element for CustomItem for GroupBy](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

NewLine Element for CustomItem for GroupBy

Article • 09/17/2021

Adds a blank line to the display of the control. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- NewLine Element

Syntax

XML

```
<NewLine/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `NewLine` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] Expand table

Element	Description
CustomItem Element for CustomEntry for GroupBy	Defines a control for the custom control view.

Remarks

See Also

[CustomItem Element for CustomEntry for GroupBy](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

⌚ [Open a documentation issue](#)

👤 [Provide product feedback](#)

Text Element for CustomItem for GroupBy

Article • 09/17/2021

Specifies text that is added to the data that is displayed by the control, such as a label, brackets to enclose the data, and spaces to indent the data. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- CustomItem Element
- Text Element

Syntax

XML

```
<Text>TextToDisplay</Text>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Text` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
CustomItem Element for CustomEntry for GroupBy	Defines a control for the custom control view.

Text Value

Specify the text of a control for data that you want to display.

Remarks

See Also

[CustomItem Element for CustomEntry for GroupBy](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

EntrySelectedBy Element for CustomEntry for GroupBy

Article • 09/17/2021

Defines the .NET types that use this control definition or the condition that must exist for this definition to be used. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element for View
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element

Syntax

XML

```
<EntrySelectedBy>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <SelectionCondition>...</SelectionCondition>
</EntrySelectedBy>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `EntrySelectedBy` element. You must specify at least one type, selection set, or selection condition for a definition. There is no maximum limit to the number of child elements that you can use.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for GroupBy	Optional element. Defines the condition that must exist for this definition to be used.
SelectionSetName Element for EntrySelectedBy for GroupBy	Optional element. Specifies a set of .NET types that use this definition of the control.
TypeName Element for EntrySelectedBy for GroupBy	Optional element. Specifies a .NET type that uses this definition of the control.

Parent Elements

[+] Expand table

Element	Description
CustomEntry Element for CustomControl for GroupBy	Provides a definition of the control.

Remarks

Selection conditions are used to define a condition that must exist for the definition to be used, such as when an object has a specific property or when a specific property value or script evaluates to `true`. For more information about selection conditions, see [Defining Conditions for when a View Entry or Item is Used](#).

See Also

[SelectionCondition Element for EntrySelectedBy for GroupBy](#)

[SelectionSetName Element for EntrySelectedBy for GroupBy](#)

[TypeName Element for EntrySelectedBy for GroupBy](#)

[CustomEntry Element for CustomEntries for Controls for View](#)

[Writing a PowerShell Formatting File](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionCondition Element for EntrySelectedBy for GroupBy

Article • 09/17/2021

Defines a condition that must exist for a control definition to be used. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element for View
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element

Syntax

XML

```
<SelectionCondition>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</SelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionCondition` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
PropertyName Element for SelectionCondition for GroupBy	Optional element. Specifies a .NET property that triggers the condition.
ScriptBlock Element for SelectionCondition for GroupBy	Optional element. Specifies the script that triggers the condition.
SelectionSetName Element for SelectionCondition for GroupBy	Optional element. Specifies the set of .NET types that triggers the condition.
TypeName Element for SelectionCondition for GroupBy	Optional element. Specifies a .NET type that triggers the condition.

Parent Elements

[+] Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for GroupBy	Defines the .NET types that use this control definition or the condition that must exist for this definition to be used.

Remarks

When you are defining a selection condition, the following requirements apply:

- The selection condition must specify at least one property name or a script block, but cannot specify both.
- The selection condition can specify any number of .NET types or selection sets, but cannot specify

both. For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

See Also

[PropertyName Element for SelectionCondition for CustomControl for View](#)

[ScriptBlock Element for SelectionCondition for CustomControl for View](#)

[SelectionSetName Element for SelectionCondition for Custom Control for View](#)

[TypeName Element for SelectionCondition for GroupBy](#)

[EntrySelectedBy Element for CustomEntry for GroupBy](#)

[Writing a PowerShell Formatting File](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



[PowerShell feedback](#)

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PropertyName Element for SelectionCondition for GroupBy

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the definition is used. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for GroupBy	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the .NET property name.

Remarks

The selection condition must specify at least one property name or a script, but cannot specify both. For more information about how selection conditions can be used, see [Defining Conditions for Displaying Data](#).

See Also

[SelectionCondition Element for EntrySelectedBy for GroupBy](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for SelectionCondition for EntrySelectedBy for GroupBy

Article • 09/17/2021

Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the definition is used. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for GroupBy	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the script that is evaluated.

Remarks

The selection condition must specify at least one script or property name to evaluate, but cannot specify both. For more information about how selection conditions can be used, see [Defining Conditions for Displaying Data](#).

See Also

[SelectionCondition Element for EntrySelectedBy for GroupBy](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for SelectionCondition for GroupBy

Article • 09/17/2021

Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met, and the object is displayed by using this control. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for GroupBy	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the name of the selection set.

Remarks

Selection sets are common groups of .NET objects that can be used by any view that the formatting file defines. For more information about creating and referencing selection sets, see [Defining Selection Sets](#).

When this element is specified, you cannot specify the [TypeName](#) element. For more information about defining selection conditions, see [Defining Conditions for Displaying Data](#).

See Also

[TypeName Element for SelectionCondition for GroupBy](#)

[SelectionCondition Element for EntrySelectedBy for GroupBy](#)

[Defining Conditions for When Data Is Displayed](#)

[Defining Selection Sets](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub



PowerShell feedback

PowerShell is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for SelectionCondition for GroupBy

Article • 09/17/2021

Specifies a .NET type that triggers the condition. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof.NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the **TypeName** Element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for GroupBy	Defines a condition that must exist for the control definition to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

When this element is specified, you cannot specify the `SelectionSetName` element. For more information about defining selection conditions, see [Defining Conditions for Displaying Data](#).

See Also

[SelectionCondition Element for EntrySelectedBy for GroupBy](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for EntrySelectedBy for GroupBy

Article • 09/17/2021

Specifies a set of .NET objects for the list entry. There is no limit to the number of selection sets that can be specified for an entry. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
EntrySelectedBy Element for CustomEntry for GroupBy	Defines the .NET types that use this custom entry or the condition that must exist for this entry to be used.

Text Value

Specify the name of the selection set.

Remarks

Each custom control definition must have at least one type name, selection set, or selection condition defined.

Selection sets are typically used when you want to define a group of objects that are used in multiple views. For example, you may want to create a table view and a list view for the same set of objects. For more information about defining selection sets, see [Defining Selection Sets](#).

For more information about the components of a custom control view, see [Creating Custom Controls](#).

See Also

[EntrySelectedBy Element for CustomEntry for GroupBy](#)

[Creating Custom Controls](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub



PowerShell feedback

PowerShell is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

TypeName Element for EntrySelectedBy for GroupBy

Article • 09/17/2021

Specifies a .NET type that uses this definition of the custom control. This element is used when defining how a new group of objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControl Element
- CustomEntries Element
- CustomEntry Element
- EntrySelectedBy Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof .NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] [Expand table](#)

Element	Description
EntrySelectedBy Element for CustomEntry for GroupBy	Defines the .NET types that use this control definition or the condition that must exist for this definition to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

Each control definition must have at least one type name, selection set, or selection condition defined.

For more information about the components of a custom control view, see [Creating Custom Controls](#).

See Also

[Creating Custom Controls](#)

[EntrySelectedBy Element for CustomEntry for GroupBy](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CustomControlName Element for GroupBy

Article • 09/17/2021

Specifies the name of a custom control that is used to display the new group. This element is used when defining a table, list, wide or custom control view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- CustomControlName Element

Syntax

XML

```
<CustomControlName>ControlName</CustomControlName>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent elements of the `CustomControlName` element.

Attributes

None.

Child Elements

None.

Parent Elements

Element	Description
GroupBy Element for View	Defines how Windows PowerShell displays a new group of objects.

Text Value

Specify the name of the custom control that is used to display a new group.

Remarks

You can create common controls that can be used by all the views of a formatting file, and you can create view controls that can be used by a specific view. The following elements specify the names of these custom controls:

- [Name Element for Control for Controls for Configuration](#)
- [Name Element for Control for Controls for View](#)

See Also

[GroupBy Element for View](#)

[Name Element for Control for Controls for Configuration](#)

[Name Element for Control for Controls for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

Label Element for GroupBy

Article • 09/17/2021

Specifies a label that is displayed when a new group is encountered.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- Label Element

Syntax

XML

```
<Label>DisplayedLabel</Label>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `Label` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
GroupBy Element for View	Defines how a new group of objects is displayed.

Text Value

Specify the text that is displayed whenever Windows PowerShell encounters a new property or script value.

Remarks

In addition to the text specified by this element, Windows PowerShell displays the new value that starts the group, and adds a blank line before and after the group.

Example

The following example shows the label for a new group. The displayed label would look similar to this: `Service Type: NewValueofProperty`

XML

```
<GroupBy>
  <Label>Service Type</Label>
  <PropertyName>ServiceType</PropertyName>
</GroupBy>
```

For an example of a complete formatting file that includes this element, see [Wide View \(GroupBy\)](#).

See Also

[GroupBy Element for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub



PowerShell feedback

PowerShell is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

PropertyName Element for GroupBy

Article • 09/17/2021

Specifies the .NET property that starts a new group whenever its value changes.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
GroupBy Element for View	Defines how a group of .NET objects is displayed.

Text Value

Specify the .NET property name.

Remarks

Windows PowerShell starts a new group whenever the value of this property changes.

When this element is specified, you cannot specify the [ScriptBlock](#) element to start a new group.

Example

The following example shows how to start a new group when the value of a property changes.

XML

```
<GroupBy>
  <Label>Service Type</Label>
  <PropertyName>ServiceType</PropertyName>
</GroupBy>
```

For an example of a complete formatting file that includes this element, see [Wide View \(GroupBy\)](#).

See Also

[GroupBy Element for View](#)

[ScriptBlock Element for GroupBy](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for GroupBy

Article • 09/17/2021

Specifies the script that starts a new group whenever its value changes.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- GroupBy Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
GroupBy Element for View	Defines how a group of .NET objects is displayed.

Text Value

Specify the script that is evaluated.

Remarks

PowerShell starts a new group whenever the value of this script changes.

When this element is specified, you cannot specify the [PropertyName](#) element to start a new group.

See Also

[PropertyName Element for GroupBy](#)

[GroupBy Element for View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

ListControl Element

Article • 09/17/2021

Defines a list format for the view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element

Syntax

XML

```
<ListControl>
  <ListEntries>...</ListEntries>
</ListControl>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `ListControl` element. This element must contain only a single child element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
ListEntries Element	Required element. Provides the definitions of the list view.

Parent Elements

[+] [Expand table](#)

Element	Description
View Element	Defines a view that is used to display the members of one or more objects.

Remarks

For more information about creating a list view, see [Creating a List View](#).

Example

This example shows a list view for the `System.ServiceProcess.ServiceController` object.

```
<View>
  <Name>System.ServiceProcess.ServiceController</Name>
  <ViewSelectedBy>
    <TypeName>System.ServiceProcess.ServiceController</TypeName>
  </ViewSelectedBy>
  <ListControl>
    <ListEntries>
      <ListEntry>...</ListEntry>
    </ListEntries>
  </ListControl>
</View>
```

See Also

[View Element](#)

[ListEntries Element](#)

[Creating a List View](#)

[Writing a Windows PowerShell Formatting and Types File](#)

ListEntries Element

Article • 03/24/2025

Provides the definitions of the list view. The list view must specify one or more definitions.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element

Syntax

XML

```
<ListEntries>
  <ListEntry>...</ListEntry>
</ListEntries>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `ListEntries` element. At least one child element must be specified.

Attributes

None.

Child Elements

 Expand table

Element	Description
ListEntry Element	Provides a definition of the list view.

Parent Elements

[+] Expand table

Element	Description
ListControl Element	Defines a list format for the view.

Remarks

For more information about list views, see [List View](#).

Example

This example shows the XML elements that define the list view for the [System.ServiceProcess.ServiceController](#) object.

XML

```
<View>
  <Name>System.ServiceProcess.ServiceController</Name>
  <ViewSelectedBy>
    <TypeName>System.ServiceProcess.ServiceController</TypeName>
  </ViewSelectedBy>
  <ListControl>
    <ListEntries>
      <ListEntry>
        <ListItems>...</ListItems>
      </ListEntry>
    </ListEntries>
  </ListControl>
</View>
```

See Also

[ListControl Element](#)

[ListEntry Element](#)

[List View](#)

[Writing a Windows PowerShell Formatting and Types File](#)

ListEntry Element

Article • 09/17/2021

Provides a definition of the list view.

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element

Syntax

XML

```
<ListEntry>
  <EntrySelectedBy>...</EntrySelectedBy>
  <ListItems>...</ListItems>
</ListEntry>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `ListEntry` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
<code>EntrySelectedBy</code> Element for <code>ListEntry</code>	Optional element. Defines the .NET objects that use this list view definition or the condition that must exist for this definition to be used.

Element	Description
ListItems Element	<p>Required element.</p> <p>Defines the properties and scripts whose values are displayed by the list view.</p>

Parent Elements

[+] Expand table

Element	Description
ListEntries Element	Provides the definitions of the list view.

Remarks

A list view is a list format that displays property values or script values for each object. For more information about list views, see [Creating a List View](#).

Example

This example shows the XML elements that define the list view for the [System.ServiceProcess.ServiceController](#) object.

XML

```
<View>
  <Name>System.ServiceProcess.ServiceController</Name>
  <ViewSelectedBy>
    <TypeName>System.ServiceProcess.ServiceController</TypeName>
  </ViewSelectedBy>
  <ListControl>
    <ListEntries>
      <ListEntry>
        <ListItems>...</ListItems>
      </ListEntry>
    </ListEntries>
  </ListControl>
</View>
```

See Also

[Creating a List View](#)

[EntrySelectedBy Element for ListEntry](#)

[ListEntries Element](#)

[ListItems Element](#)

[Writing a Windows PowerShell Formatting and Types File](#)

EntrySelectedBy Element for ListEntry

Article • 09/17/2021

Defines the .NET types that use this list view definition or the condition that must exist for this definition to be used. In most cases only one definition is needed for a list view. However, you can provide multiple definitions for the list view if you want to use the same list view to display different data for different objects.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- EntrySelectedBy Element

Syntax

XML

```
<EntrySelectedBy>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <SelectionCondition>...</SelectionCondition>
</EntrySelectedBy>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `EntrySelectedBy` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for ListControl	Optional element. Defines the condition that must exist for this list view definition to be used.
SelectionSetName Element for EntrySelectedBy for ListControl	Optional element. Specifies a set of .NET types that use this list view definition.
TypeName Element for EntrySelectedBy for ListControl	Optional element. Specifies a .NET type that uses this list view definition.

Parent Elements

[+] Expand table

Element	Description
ListEntry Element for ListControl	Defines how the rows of the list are displayed.

Remarks

You must specify at least one type, selection set, or selection condition for a list view definition. There is no maximum limit to the number of child elements that you can use.

Selection conditions are used to define a condition that must exist for the definition to be used, such as when an object has a specific property or that a specific property value or script evaluates to `true`. For more information about selection conditions, see [Defining Conditions for when Data is displayed](#).

For more information about the components of a list view, see [Creating a List View](#).

Example

The following example shows how to define the objects for a list view using their .NET type name.

XML

```
<ListEntry>
  <EntrySelectedBy>
    <TypeName>NameofDotNetType</TypeName>
  </EntrySelectedBy>
</ListEntry>
```

See Also

[ListEntry Element for ListControl](#)

[SelectionCondition Element for EntrySelectedBy for ListControl](#)

[SelectionSetName Element for EntrySelectedBy for ListControl](#)

[TypeName Element for EntrySelectedBy for ListControl](#)

[Creating a List View](#)

[Defining Conditions for when Data is Displayed](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionCondition Element for EntrySelectedBy for ListControl

Article • 09/17/2021

Defines the condition that must exist to use this definition of the list view. There is no limit to the number of selection conditions that can be specified for a list definition.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- EntrySelectedBy Element
- SelectionCondition Element

Syntax

XML

```
<SelectionCondition>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</SelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionCondition` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
PropertyName Element for SelectionCondition for EntrySelectedBy for ListEntry	Optional element. Specifies the .NET property that triggers the condition.
ScriptBlock Element for SelectionCondition for EntrySelectedBy for ListEntry	Optional element. Specifies the script that triggers the condition.
SelectionSetName Element for SelectionCondition for EntrySelectedBy for ListEntry	Optional element. Specifies the set of .NET types that trigger the condition.
TypeName Element for SelectionCondition for EntrySelectedBy for ListEntry	Optional element. Specifies a .NET type that triggers the condition.

Parent Elements

[+] Expand table

Element	Description
EntrySelectedBy Element for TableRowEntry	Defines the .NET types that use this table entry or the condition that must exist for this entry to be used.

Remarks

When you are defining a selection condition, the following requirements apply:

- The selection condition must specify at least one property name or a script block, but cannot specify both.
- The selection condition can specify any number of .NET types or selection sets, but cannot specify both.

For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

For more information about other components of a list view, see [Creating a List View](#).

See Also

[Creating a List View](#)

[Defining Conditions for When Data Is Displayed](#)

[ListEntry Element](#)

[SelectionSetName Element for EntrySelectedBy for ListEntry](#)

[TypeName Element for EntrySelectedBy for ListEntry](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

PropertyName Element for SelectionCondition for EntrySelectedBy for ListControl

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the list entry is used.

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for ListEntry	Defines the condition that must exist for this list entry to be used.

Text Value

Specify the .NET property name.

Remarks

The selection condition must specify at least one property name or a script block, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when a View Entry or Item is Used](#).

For more information about other components of a list view, see [Creating List View](#).

See Also

[Creating a List View](#)

[Defining Conditions for When Data is Displayed](#)

[ListEntry Element](#)

[ScriptBlock Element for SelectionCondition for EntrySelectedBy for ListEntry](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

ScriptBlock Element for SelectionCondition for EntrySelectedBy for ListControl

Article • 09/17/2021

Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the list entry is used.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for ListEntry	Defines the condition that must exist for this list entry to be used.

Text Value

Specify the script that is evaluated.

Remarks

The selection condition must specify at least one script or property name to evaluate, but cannot specify both. (For more information about how selection conditions can be used, see [Defining Conditions for when a View Entry or Item is Used](#).)

For more information about the other components of a list view, see [List View](#).

See Also

[ListEntry Element](#)

[PropertyName Element for SelectionCondition for EntrySelectedBy for ListEntry](#)

[SelectionCondition Element for EntrySelectedBy for ListEntry](#)

[List View](#)

[Defining Conditions for when a View Entry or Item is Used](#)

[Writing a Windows PowerShell Formatting and Types File](#)



[PowerShell feedback](#)

PowerShell is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for SelectionCondition for EntrySelectedBy for ListEntry

Article • 09/17/2021

Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met, and the object is displayed by using this definition of the list view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for ListEntry	Defines the condition that must exist to use this definition of the list view.

Text Value

Specify the name of the selection set.

Remarks

The selection condition can specify a selection set or .NET type, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

Selection sets are common groups of .NET objects that can be used by any view that the formatting file defines. For more information about creating and referencing selection sets, see [Defining Sets of Objects](#).

For more information about other components of a list view, see [Creating a List View](#).

See Also

[Creating a List View](#)

[Defining Conditions for When Data Is Displayed](#)

[SelectionCondition Element for EntrySelectedBy for ListEntry](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for SelectionCondition for EntrySelectedBy for ListControl

Article • 09/17/2021

Specifies a .NET type that triggers the condition. When this type is present, the list entry is used.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof .NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for ListControl	Defines the condition that must exist for this list entry to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

The selection condition can specify any number of .NET types or selection sets, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

For more information about other the components of a list view, see [Creating a List View](#).

See Also

[Creating a List View](#)

[Defining Conditions for When Data Is Displayed](#)

[SelectionCondition Element for EntrySelectedBy for ListControl](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



[PowerShell feedback](#)

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

SelectionSetName Element for EntrySelectedBy for ListControl

Article • 09/17/2021

Specifies a set of .NET objects for the list entry. There is no limit to the number of selection sets that can be specified for an entry.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- EntrySelectedBy Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
EntrySelectedBy Element for ListEntry	Defines the .NET types that use this list entry or the condition that must exist for this entry to be used.

Text Value

Specify the name of the selection set.

Remarks

Each list entry must have at least one type name, selection set, or selection condition defined.

Selection sets are typically used when you want to define a group of objects that are used in multiple views. For example, you might want to create a table view and a list view for the same set of objects. For more information about defining selection sets, see [Defining Sets of objects for a View](#).

For more information about the components of a list view, see [Creating a List View](#).

Example

The following example shows how to specify a selection set for an entry of a list view.

XML

```
<ListEntry>
  <EntrySelectedBy>
    <SelectionSetName>NameofSelectionSet</SelectionSetName>
  </EntrySelectedBy>
  <ListItems>...</ListItems>
</ListEntry>
```

See Also

[Creating a List View](#)

EntrySelectedBy Element for ListEntry

Writing a PowerShell Formatting File

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for EntrySelectedBy for ListControl

Article • 09/17/2021

Specifies a .NET type that uses this entry of the list view. There is no limit to the number of types that can be specified for a list entry.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- EntrySelectedBy
- TypeName Element

Syntax

XML

```
<TypeName>Nameof .NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] [Expand table](#)

Element	Description
EntrySelectedBy Element for ListEntry	Defines the .NET types that use this list entry or the condition that must exist for this entry to be used.

Text Value

Specify the fully-qualified name of the .NET type, such as `System.IO DirectoryInfo`.

Remarks

Each list entry must have at least one type name, selection set, or selection condition defined.

For more information about how this element is used in a list view, see [List View](#).

Example

The following example shows how to specify a selection set for an entry of a list view.

XML

```
<ListEntry>
  <EntrySelectedBy>
    <TypeName>Nameof.NetType</TypeName>
  </EntrySelectedBy>
  <ListItems>...</ListItems>
</ListEntry>
```

See Also

[Creating a List View](#)

[EntrySelectedBy Element for ListEntry](#)

[SelectionSetName Element for EntrySelectedBy for ListEntry](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ListItems Element

Article • 09/17/2021

Defines the properties and scripts whose values are displayed in the rows of the list view.

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- ListItems Element

Syntax

XML

```
<ListItems>
  <ListItem>...</ListItem>
</ListItems>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `ListItems` element. There is no limit to the number of child elements that can be specified. The order of the child elements defines the order that values are displayed in the list view.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
ListItem Element for ListControl	Required element. Defines the property or script whose value is displayed by the list view.

Parent Elements

[\[+\] Expand table](#)

Element	Description
ListEntry Element for ListControl	Provides a definition of the list view.

Remarks

For more information about this type of view, see [Creating a List View](#).

Example

This example shows the XML elements that define three rows of the list view.

XML

```
<ListEntry>
  <ListItems>
    <ListItem>
      <Label>Property1: </Label>
      <PropertyName>.NetTypeProperty1</PropertyName>
    </ListItem>
    <ListItem>
      <PropertyName>.NetTypeProperty2</PropertyName>
    </ListItem>
    <ListItem>
      <ScriptBlock>$_.ProcessName + ":" $_.Id</ScriptBlock>
    </ListItem>
  </ListItems>
</ListEntry>
```

See Also

[ListEntry Element for ListControl](#)

[ListItem Element for ListControl](#)

Creating a List View

Writing a PowerShell Formatting File

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

FormatString Element for ListItem for ListControl

Article • 01/18/2022

Specifies a format pattern that defines how the property or script value is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- ListItems Element
- ListItem Element
- FormatString Element

Syntax

XML

```
<FormatString>PropertyPattern</FormatString>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `FormatString` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
ListItem Element	Defines the property or script whose value is displayed in a row of the list view.

Text Value

Specify the pattern that is used to format the data. For example, you can use this pattern to format the value of any property that is of type [System.TimeSpan](#): {0:MMM}{0:dd}{0:HH}:{0:mm}.

Remarks

Format strings can be used when creating table views, list views, wide views, or custom views. For more information about formatting a value displayed in a view, see [Formatting Displayed Data](#).

For more information about using format strings in list views, see [Creating List View](#).

Example

The following example shows how to define a formatting string for the value of the `StartTime` property.

XML

```
<ListItem>
  <PropertyName>StartTime</PropertyName>
  <FormatString>{0:MMM} {0:DD} {0:HH}:{0:MM}</FormatString>
</ListItem>
```

See Also

[Creating a List View](#)

[ListItem Element](#)

[Writing a Windows PowerShell Formatting and Types File](#)

ItemSelectionCondition Element for ListItem for ListControl

Article • 09/17/2021

Defines the condition that must exist for this list item to be used.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- ListItems Element
- ListItem Element
- ItemSelectionCondition Element

Syntax

XML

```
<ItemSelectionCondition>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</ItemSelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ItemSelectionCondition` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
PropertyName Element for ItemSelectionCondition for ListControl	Optional element. Specifies the .NET property that triggers the condition.
ScriptBlock Element for ItemSelectionCondition for ListControl	Optional element. Specifies the script that triggers the condition.

Parent Elements

[+] Expand table

Element	Description
ListItem Element for ListItems for ListControl	Defines the property or script whose value is displayed in a row of the list view.

Remarks

You can specify one property name or a script for this condition but cannot specify both.

See Also

[ListItem Element for ListItems for ListControl](#)

[PropertyName Element for ItemSelectionCondition for ListControl](#)

[ScriptBlock Element for ItemSelectionCondition for ListControl](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

PropertyName Element for ItemSelectionCondition for ListControl

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the view is used. This element is used when defining a list view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- ListItems Element
- ListItem Element
- ItemSelectionCondition Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent elements of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ItemSelectionCondition Element for ListItem for ListControl	

Text Value

Specify the name of the property whose value is displayed.

Remarks

If this element is used, you cannot specify the [ScriptBlock](#) element when defining the selection condition.

See Also

[ScriptBlock Element for ItemSelectionCondition for ListControl](#)

[ItemSelectionCondition Element for ListItem for ListControl](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for ItemSelectionCondition for ListControl

Article • 09/17/2021

Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the list item is used. This element is used when defining a list view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element for ListControl
- ListEntry Element for ListEntries for ListControl
- ListItems Element for ListEntry for ListControl
- ListItem Element for ListItems for List Control
- ItemSelectionCondition Element for ListItem for ListControl
- ScriptBlock Element for ItemSelectionCondition for ListControl

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent elements of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ItemSelectionCondition Element for ListItem for ListControl	Defines the condition that must exist for this list item to be used.

Text Value

Specify the script that is evaluated.

Remarks

If this element is used, you cannot specify the `PropertyName` element when defining the selection condition.

See Also

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Label Element for ListItem for ListControl

Article • 09/17/2021

Specifies the label that is displayed to the left of the property or script value in the row.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- ListItems Element
- ListItem Element
- Label Element

Syntax

XML

```
<Label>Label for displayed value</Label>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `Label` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] [Expand table](#)

Element	Description
ListItem Element for ListItems for ListControl	Defines the property or script whose value is displayed in a row of the list view.

Text Value

Specify the label to be display to the left of the property or script value.

Remarks

If a label is not specified, the name of the property or the script is displayed. For more information about using labels in a list view, see [Creating a List View](#).

Example

The following example shows how to add a label to a row.

XML

```
<ListItem>
  <Label>Property1: </Label>
  <PropertyName>DotNetProperty1</PropertyName>
</ListItem>
```

See Also

[Creating a List View](#)

[ListItem Element](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on



PowerShell feedback

GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ListItem Element

Article • 09/17/2021

Defines the property or script whose value is displayed in a row of the list view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- ListItems Element
- ListItem Element

Syntax

XML

```
<ListItem>
  <PropertyName>PropertyToDisplay</PropertyName>
  <ScriptBlock>ScriptToExecute</ScriptBlock>
  <Label>LabelToDisplay</Label>
  <FormatString>FormatPattern</FormatString>
  <ItemSelectionCondition>...</ItemSelectionCondition>
</ListItem>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `ListItem` element. Only one property or script can be specified.

Attributes

None

Child Elements

[] Expand table

Element	Description
FormatString Element for ListItem for ListControl	Optional element. Specifies a format string that defines how the property or script value is displayed.
ItemSelectionCondition Element for ListItem for ListControl	Optional element. Defines the condition that must exist for this list item to be used.
Label Element for ListItem for ListControl	Optional element Specifies the label that is displayed to the left of the property or script value in the row.
PropertyName Element for ListItem for ListControl	Optional element. Specifies the .NET property whose value is displayed in the row.
ScriptBlock Element for ListItem for ListControl	Optional element. Specifies the script whose value is displayed in the row.

Parent Elements

[] Expand table

Element	Description
ListItems Element for List Control	Defines the properties and scripts whose values are displayed in the list view.

Remarks

For more information about the components of a list view, see [Creating a List View](#).

Example

This example shows the XML elements that define three rows of the list view. The first two rows display the value of a .NET property, and the last row displays a value generated by a script.

```
XML

<ListEntry>
  <ListItems>
    <ListItem>
      <Label>Property1: </Label>
      <PropertyName>DotNetProperty1</PropertyName>
    </ListItem>
    <ListItem>
      <PropertyName>DotNetProperty2</PropertyName>
    </ListItem>
    <ListItem>
      <ScriptBlock>$_.ProcessName + ":" $_.Id</ScriptBlock>
    </ListItem>
  </ListItems>
</ListEntry>
```

See Also

[ListItems Element](#)

[FormatString Element for ListItem](#)

[Label Element for ListItem](#)

[PropertyName Element for ListItem](#)

[ScriptBlock Element for ListItem](#)

[Creating a List View](#)

[Writing a Windows PowerShell Formatting and Types File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review
issues and pull requests. For



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:



[Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

PropertyName Element for ListItem for ListControl

Article • 09/17/2021

Specifies the .NET property whose value is displayed in the list.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- ListItems Element
- ListItem Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
ListItem Element	Defines the property or script whose value is displayed in the row of the list view.

Text Value

Specify the name of the property whose value is displayed.

Remarks

When this element is specified, you cannot specify the [ScriptBlock](#) element.

In addition to displaying the property value, you can also specify a label for the value or a format string that can be used to change the display of the value. For more information about specifying data in a list view, see [Creating a List View](#).

Example

The following example shows how to specify the label and property whose value is displayed.

XML

```
ListItem>
  <Label>NameOfProperty</Label>
  <PropertyName>.NetTypeProperty</PropertyName>
</ListItem>
```

See Also

[ScriptBlock Element for ListItem for ListControl](#)

[Creating a List View](#)

[ListItem Element for ListControl\(Format\)](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for ListItem for ListControl

Article • 09/17/2021

Specifies the script whose value is displayed in the row.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ListControl Element
- ListEntries Element
- ListEntry Element
- ListItems Element
- ListItem Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
ListItem Element	Defines the property or script whose value is displayed in a row of the list view.

Text Value

Specify the script whose value is displayed in the row.

Remarks

When this element is specified, you cannot specify the [PropertyName](#) element.

For more information about specifying scripts in a list view, see [List View](#).

Example

The following example shows how to specify the property whose value is displayed.

XML

```
<ListItem>
  <ScriptBlock>$_.ProcessName + ":" $_.Id</ScriptBlock>
</ListItem>
```

See Also

[PropertyName Element for ListItem for ListControl](#)

[Creating a List View](#)

[ListItem Element for ListItems for ListControl](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Name Element for View

Article • 03/24/2025

Specifies the name that is used to identify the view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- Name Element

Syntax

XML

```
<Name>ViewName</Name>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `Name` element. Only one `Name` element is allowed for each view.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
View Element	Defines a view that is used to display the members of one or more .NET objects.

Text Value

Specify a unique friendly name for the view. This name can include a reference to the type of the view (such as a table view or list view), which object or set of objects use the view, what command returns the objects, or a combination of these.

Remarks

For more information about the different types of views, see the following topics: [Table View](#), [List View](#), [Wide View](#), and [Custom View](#).

Example

The following example shows a `View` element that defines a table view for the `System.ServiceProcess.ServiceController` object. The name of the view is "service".

```
XML

<View>
  <Name>service</Name>
  <ViewSelectedBy>
    <TypeName>System.ServiceProcess.ServiceController</TypeName>
  </ViewSelectedBy>
  <TableControl>...</TableControl>
</View>
```

See Also

[Creating a List View](#)

[Creating a Table View](#)

[Creating a Wide View](#)

[Creating Custom Controls](#)

[View Element](#)

[Writing a PowerShell Formatting File](#)

OutOfBand Element

Article • 09/17/2021

In a pipeline, the first object emitted is chosen as the type to format the output of the pipeline. PowerShell attempts to format subsequent objects using the same view. If the object does not fit the view, it is not displayed. You can create OutOfBand views that can be used to format these other types.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- OutOfBand Element

Syntax

XML

```
<OutOfBand/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `OutOfBand` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] Expand table

Element	Description
View Element	Defines a view that displays one or more .NET objects.

Remarks

When the "shape" of formatting (view) has been determined by previous objects, you may want objects of different types to continue using that shape (table, list, or whatever) even if they specify their own views. Or sometimes you want your view to take over. When `OutOfBand` is true, the view applies regardless of previous objects that may have selected a different view.

See Also

[View Element](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TableControl Element

Article • 09/17/2021

Defines a table format for a view.

Schema

- ViewDefinitions Element
- View Element
- TableControl Element

Syntax

XML

```
<TableControl>
  <AutoSize/>
  <HideTableHeaders/>
  <TableHeaders>...</TableHeaders>
  <TableRowEntries>...</TableRowEntries>
</TableControl>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `TableControl` element. You must specify the rows of the table. All other child elements are optional.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
AutoSize Element for TableControl	Optional element. Specifies whether the column size and the number of columns are adjusted based on the size of the data.
HideTableHeaders Element for TableControl	Optional element. Indicates whether the header of the table is not displayed.
TableHeaders Element for TableControl	Required element. Defines the labels, the widths, and the alignment of the data for the columns of the table view.
TableRowEntries Element for TableControl	Optional element. Provides the definitions of the table view.

Parent Elements

[\[\] Expand table](#)

Element	Description
View Element	Defines a view that is used to display the members of one or more objects.

Remarks

For more information about the components of a table view, see [Creating a Table View](#).

Example

This example shows a `TableControl` element that is used to display the properties of the `System.ServiceProcess.ServiceController` object.

XML

```
<View>
  <Name>service</Name>
  <ViewSelectedBy>
    <TypeName>System.ServiceProcess.ServiceController</TypeName>
  </ViewSelectedBy>
  <TableControl>
```

```
<TableHeaders>...</TableHeaders>
<TableRowEntries>...</TableRowEntries>
</TableControl>
</View>
```

See Also

[Creating a Table View](#)

[View Element](#)

[AutoSize Element for TableControl](#)

[HideTableHeaders Element](#)

[TableHeaders Element](#)

[TableRowEntries Element](#)

[Writing a PowerShell Formatting File](#)

AutoSize Element for TableControl

Article • 09/17/2021

Specifies whether the column size and the number of columns are adjusted based on the size of the data.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- AutoSize Element

Syntax

XML

```
<AutoSize/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `AutoSize` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
TableControl Element	Defines a table format for a view.

Remarks

For more information about the components of a table view, see [Creating a Table View](#).

See Also

[Creating a Table View](#)

[TableControl Element](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

HideTableHeaders Element

Article • 09/17/2021

Specifies that the headers of the table are not displayed.

Schema

- ViewDefinitions Element
- View Element
- TableControl Element
- HideTableHeaders Element

Syntax

VB

```
<HideTableHeaders/>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `HideTableHeaders` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
TableControl Element	Defines a table format for a view.

Text Value

Specify `true` to hide the headers of the table.

Remarks

For more information about the components of a table view, see [Creating a Table View](#).

See Also

[Creating a Table View](#)

[TableControl Element](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TableHeaders Element

Article • 09/17/2021

Defines the headers for the columns of a table.

Schema

- ViewDefinitions Element
- View Element
- TableControl Element
- TableHeaders Element for TableControl

Syntax

XML

```
<TableHeaders>
  <TableColumnHeader>...</TableColumnHeader>
</TableHeaders>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent elements of the `TableHeaders` element. There must be a child element for each property of the object that is to be displayed. The column header information is displayed in the order that the child elements are specified.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
TableColumnHeader Element	Defines the label, the width, and the alignment of the data for a column of a table view.

Parent Elements

[\[+\] Expand table](#)

Element	Description
TableControl Element	Defines a table format for a view.

Remarks

For more information about the components of a table view, see [Creating a Table View](#).

Example

This example shows a `TableHeaders` element that defines two column headers.

XML
<pre><TableHeaders> <TableColumnHeader> <Label>Column 1</Label> <Width>16</Width> <Alignment>Left</Alignment> </TableColumnHeader> <TableColumnHeader> <Label>Column 2</Label> <Width>10</Width> <Alignment>Centered</Alignment> </TableColumnHeader> </TableHeaders></pre>

See Also

[Creating a Table View](#)

[TableColumnHeader Element](#)

TableControl Element

Writing a PowerShell Formatting File

TableColumnHeader Element

Article • 09/17/2021

Defines the label, the width of the column, and the alignment of the label for a column of the table.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableHeaders Element
- TableColumnHeader Element

Syntax

XML

```
<TableColumnHeader>
  <Label>DisplayedLabel</Label>
  <Width>NumberOfCharacters</Width>
  <Alignment>Left, Right, or Centered</Alignment>
</TableColumnHeader>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TableColumnHeader` element.

Attributes

None.

Child Elements

 Expand table

Element	Description
Label Element For TableColumnHeader for TableControl	Optional element. Defines the label that is displayed at the top of the column. If no label is specified, the name of the property whose value is displayed in the rows is used.
Width Element for TableColumnHeader for TableControl	Required element. Specifies the width (in characters) of the column.
Alignment Element for TableColumnHeader for TableControl	Optional element. Specifies how the label of the column is displayed. If no alignment is specified, the label is aligned on the left.

Parent Elements

[\[\] Expand table](#)

Element	Description
TableHeaders Element	Defines the columns of a table view.

Remarks

Specify a header for each column of the table. The columns are displayed in the order in which the `TableColumnHeader` elements are defined.

A table must have the same number of `TableColumnHeader` elements as `TableRowEntry` elements. The column header defines how the text at the top of the table is displayed. The row entries define what data is displayed in the rows of the table.

For more information about the components of a table view, see [Table View](#).

Example

The following example shows two `TableColumnHeader` elements. The first element defines a column whose label is "Column 1", has a width of 16 characters, and whose label is aligned on the left. The second element defines a column whose label is "Column 2", has a width of 10 characters, and whose label is centered in the column.

XML

```
<TableHeaders>
  <TableColumnHeader>
    <Label>Column 1</Label>
    <Width>16</Width>
    <Alignment>Left</Alignment>
  </TableColumnHeader>
  <TableColumnHeader>
    <Label>Column 2</Label>
    <Width>10</Width>
    <Alignment>Centered</Alignment>
  </TableColumnHeader>
</TableHeaders>
```

See Also

[Alignment Element for TableColumnHeader for TableControl](#)

[Creating a Table View](#)

[Label Element for TableColumnHeader for TableControl](#)

[TableHeaders Element for TableControl](#)

[Width for TableColumnHeader for TableControl Element](#)

[Writing a PowerShell Formatting File](#)

Alignment Element for TableColumnHeader

Article • 09/17/2021

Defines how the data in a column header is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableHeaders Element
- TableColumnHeader Element
- Alignment Element

Syntax

XML

```
<Alignment>AlignmentType</Alignment>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `Alignment` element.

Attributes

None.

Child Elements

None.

Parent Elements

Element	Description
TableColumnHeader Element	Defines a label, the width, and the alignment of the data for a column of the table.

Text Value

Specify one of the following values. These values are not case-sensitive.

- Left - Aligns the data displayed in the column on the left. This is the default if this element is not specified.
- Right - Aligns the data displayed in the column on the right.
- Center - Centers the data displayed in the column.

Remarks

For more information about the components of a table view, see [Creating a Table View](#).

Example

This example shows a `TableColumnHeader` element whose data is aligned on the center.

XML

```
<TableColumnHeader>
  <Label>Column 1</Label>
  <Width>16</Width>
  <Alignment>Center</Alignment>
</TableColumnHeader>
```

See Also

[Creating a Table View](#)

[TableColumnHeader Element](#)

[Writing a PowerShell Formatting File](#)

Label Element for TableColumnHeader for TableControl

Article • 09/17/2021

Defines the label that is displayed at the top of a column. This element is used when defining a table view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableHeaders Element
- TableColumnHeader Element
- Label Element

Syntax

XML

```
<Label>DisplayedLabel</Label>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `Label` element. Only one label is allowed for each column.

Attributes

None.

Child Elements

None.

Parent Elements

[] [Expand table](#)

Element	Description
TableColumnHeader Element for TableHeaders for TableControl	Defines a label, the width, and the alignment of the data for a column of the table.

Text Value

Specify the text that is displayed at the top of the column of the table. There are no restricted characters for the column label.

Remarks

If no label is specified, the name of the property whose value is displayed in the rows is used.

For more information about the components of a table view, see [Creating a Table View](#).

Example

This example shows a `TableColumnHeader` element whose label is "Column 1".

XML

```
<TableColumnHeader>
  <Label>Column 1</Label>
  <Width>16</Width>
  <Alignment>Left</Alignment>
</TableColumnHeader>
```

See Also

[Creating a Table View](#)

[TableColumnHeader Element](#)

[Writing a PowerShell Formatting File](#)

Width Element for TableColumnHeader

Article • 09/17/2021

Defines the width (in characters) of a column.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableHeaders Element
- TableColumnHeader
- Width Element

Syntax

XML

```
<Width>NumberOfCharacters</Width>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `Width` element used when defining column headers.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
TableColumnHeader Element for TableHeaders for TableControl	Defines a label, width, and alignment of the data for a column of the table.

Text Value

When at all possible, specify a width (in characters) that is greater than the length of the displayed property values.

Remarks

For more information about the components of a table view, see [Creating a Table View](#).

Example

The following example shows a `TableColumnHeader` element whose width is 16 characters.

XML

```
<TableColumnHeader>
  <Label>Column 1</Label>
  <Width>16</Width>
  <Alignment>Left</Alignment>
</TableColumnHeader>
```

See Also

[Creating a Table View](#)

[TableColumnHeader Element for TableHeader for TableControl](#)

[Writing a PowerShell Formatting File](#)

TableRowEntries Element

Article • 09/17/2021

Defines the rows of the table.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element

Syntax

XML

```
<TableRowEntries>
  <TableRowEntry>...</TableRowEntry>
</TableRowEntries>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `TableRowEntries` element.

Attributes

None.

Child Elements

[] Expand table

Element	Description
TableRowEntry Element for TableRowEntries for TableControl	Required element.

Element	Description
	Defines the data that is displayed in a row of the table.

Parent Elements

[\[+\] Expand table](#)

Element	Description
TableControl Element	Defines a table format for a view.

Remarks

You must specify one or more `TableRowEntry` elements for the table view. There is no maximum limit to the number of `TableRowEntry` elements that can be added nor is their order significant.

For more information about the components of a table view, see [Creating a Table View](#).

Example

The following example shows a `TableRowEntries` element that defines a row that displays the values of two properties of the `System.Diagnostics.Process` object.

XML
<pre><TableRowEntries> <TableRowEntry> <EntrySelectedBy> <TypeName>System.Diagnostics.Process</TypeName> </EntrySelectedBy> <TableColumnItems> < TableColumnItem> <PropertyName> Property for first column</PropertyName> </ TableColumnItem> < TableColumnItem> <PropertyName> Property for second column</PropertyName> </ TableColumnItem> </TableColumnItems> </TableRowEntry> </TableRowEntries></pre>

See Also

[Creating a Table View](#)

[TableControl Element](#)

[TableRowEntry Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

TableColumnItems Element

Article • 02/03/2023

Defines the properties or scripts whose values are displayed in a row.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element for TableControl
- TableRowEntry Element for TableRowEntries for TableControl
- TableColumnItems Element for TableControlEntry for TableControl

Syntax

XML

```
<TableColumnItems>
  <TableColumnItem>...</TableColumnItem>
</TableColumnItems>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `TableColumnItems` element.

Attributes

None.

Child Elements

 Expand table

Element	Description
TableColumnItem Element for TableColumnItems for TableControl	Required element. Defines the property or script whose value is displayed in a column of the row.

Parent Elements

 Expand table

Element	Description
TableRowEntry Element for TableRowEntries for TableControl	Defines the data that is displayed in a row of the table.

Remarks

A `TableColumnItem` element is required for each column of the row. The first entry is displayed in first column, the second entry in the second column, and so on.

For more information about the components of a table view, see [Creating a Table View](#).

Example

The following example shows a `TableColumnItems` element that defines three properties of the `System.Diagnostics.Process` object.

XML

```
<TableColumnItems>
  < TableColumnItem>
    <PropertyName>Status</PropertyName>
  </ TableColumnItem>
  < TableColumnItem>
    <PropertyName>Name</PropertyName>
  </ TableColumnItem>
  < TableColumnItem>
    <PropertyName>DisplayName</PropertyName>
  </ TableColumnItem>
</TableColumnItems>
```

See Also

[Creating a Table View](#)

[TableColumnItem Element](#)

[TableRowEntry Element](#)

[Writing a PowerShell Formatting File](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

TableColumnItem Element

Article • 09/17/2021

Defines the property or script whose value is displayed in the column of the row.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- TableColumnItems Element
- TableColumnItem Element

Syntax

XML

```
<TableColumnItem>
  <Alignment>Left, Right, or Center</Alignment>
  <FormatString>FormatPattern</FormatString>
  <PropertyName>Nameof.NetProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</TableColumnItem>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `TableColumnItem` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
Alignment Element for TableColumnItem for TableControl	Optional element. Defines how the data in a column of the row is displayed.
FormatString Element for TableColumnItem for TableControl	Specifies a format pattern that is used to format the data in the column of the row.
PropertyName Element for TableColumnItem for TableControl	Optional element. Specifies the name of the property whose value is displayed.
ScriptBlock Element for TableColumnItem for TableControl	Optional element. Specifies the script whose value is displayed in the column of a row.

Parent Elements

[+] Expand table

Element	Description
TableColumnItems Element for TableControlEntry for TableControl	Defines the properties or scripts whose values are displayed in the row.

Remarks

You can specify a property of an object or a script in each column of the row. If no child elements are specified, the item is a placeholder, and no data is displayed.

For more information about the components of a table view, see [Creating a Table View](#).

Example

This example shows a `TableColumnItem` element that displays the value of the `Status` property of the `System.Diagnostics.Process` object.

XML

```
<TableColumnItem>
  <Alignment>Centered</Alignment>
  <PropertyName>Status</PropertyName>
</TableColumnItem>
```

See Also

[Creating a Table View](#)

[Alignment Element for TableColumnItem for TableControl](#)

[TableColumnItems Element](#)

[FormatString Element for TableColumnItem for TableControl](#)

[PropertyName Element for TableColumnItem for TableControl](#)

[ScriptBlock Element for TableColumnItem for TableControl](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

Alignment Element for TableColumnItem

Article • 09/17/2021

Defines how the data in a column of the row is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- TableColumnItems Element
- TableColumnItem Element
- Alignment Element

Syntax

XML

```
<Alignment>AlignmentType</Alignment>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `Alignment` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
TableColumnItem Element	Defines a label, the width, and the alignment of the data for a column of the table.

Text Value

Specify one of the following values. (These values are not case-sensitive.)

- Left - Shifts the data displayed in the column to the left. (This is the default if this element is not specified.)
- Right - Shifts the data displayed in the column to the right.
- Center - Centers the data displayed in the column.

Remarks

For more information about the components of a table view, see [Table View](#).

See Also

[Table View](#)

[TableColumnItem Element](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

FormatString Element for TableColumnItem for TableControl

Article • 01/18/2022

Specifies a format pattern that defines how the property or script value of the table is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- TableColumnItems Element
- TableColumnItem Element
- FormatString Element

Syntax

XML

```
<FormatString>FormatPattern</FormatString>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `FormatString` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
TableColumnItem Element	Defines the property or script whose value is displayed in the column of the row.

Text Value

Specify the pattern that is used to format the data. For example, this pattern can be used to format the value of any property that is of type [System.TimeSpan](#): {0:MMM}{0:dd}{0:HH}:{0:mm}.

Remarks

Format strings can be used when creating table views, list views, wide views, or custom views. For more information about formatting a value displayed in a view, see [Formatting Displayed Data](#).

For more information about the components of a table view, see [Table View](#).

Example

The following example shows how to define a formatting string for the value of the `StartTime` property.

XML

```
<TableColumnItem>
  <PropertyName>StartTime</PropertyName>
  <FormatString>{0:MMM} {0:DD} {0:HH}:{0:MM}</FormatString>
</TableColumnItem>
```

See Also

[Creating a Table View](#)

[Formatting Displayed Data](#)

TableColumnItem Element

Writing a PowerShell Formatting File

PropertyName Element for TableColumnItem for TableControl

Article • 09/17/2021

Specifies the property whose value is displayed in the column of the row.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- TableColumnItems Element
- TableColumnItem Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
TableColumnItem Element	Defines the property or script whose value is displayed in the column of the row.

Text Value

Specify the name of the property whose value is displayed.

Remarks

For more information about the components of a table view, see [Creating a Table View](#).

Example

This example shows a `TableColumnItem` element that specifies the `Status` property of the `System.Diagnostics.Process` object.

XML

```
<TableColumnItem>
  <Alignment>Centered</Alignment>
  <PropertyName>Status</PropertyName>
</TableColumnItem>
```

See Also

[Creating a Table View](#)

[TableColumnItem Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for TableColumnItem for TableControl

Article • 09/17/2021

Specifies the script whose value is displayed in the column of the row.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- TableColumnItems Element
- TableColumnItem Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] Expand table

Element	Description
TableColumnItem Element	Defines the property or script whose value is displayed in the column of the row.

Text Value

Specify the script whose value is displayed.

Remarks

For more information about the components of a table view, see [Creating a Table View](#).

See Also

[Creating a Table View](#)

[TableColumnItem Element](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TableRowEntry Element

Article • 09/17/2021

Defines the data that is displayed in a row of the table.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element for TableControl
- TableRowEntry Element for TableRowEntries

Syntax

XML

```
<TableRowEntry>
  <Wrap/>
  <EntrySelectedBy>...</EntrySelectedBy>
  <TableColumnItems>...</TableColumnItems>
</TableRowEntry>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent element of the `TableRowEntry` element.

Attributes

None.

Child Elements

 Expand table

Element	Description
EntrySelectedBy Element for TableRowEntry for TableControl	Required element. Defines the objects whose property values are displayed in the row.
TableColumnItems Element for TableRowEntry for TableControl	Required element. Defines the properties or scripts whose values are displayed.
Wrap Element for TableRowEntry for TableControl	Optional element. Specifies that text that exceeds the column width is displayed on the next line.

Parent Elements

[\[\] Expand table](#)

Element	Description
TableRowEntries Element for TableControl	Defines the rows of the table.

Remarks

One [TableColumnItems](#) element and one [EntrySelectedBy](#) element must be specified.

For more information about the components of a table view, see [Creating a Table View](#).

Example

The following example shows a [TableRowEntry](#) element that defines a row that displays the values of two properties of the [System.Diagnostics.Process](#) object.

XML

```

<TableRowEntry>
  <EntrySelectedBy>
    <TypeName>System.Diagnostics.Process</TypeName>
  </EntrySelectedBy>
  <TableColumnItems>
    < TableColumnItem>
      <PropertyName> Property for first column</PropertyName>
    </ TableColumnItem>
  </TableColumnItems>
</TableRowEntry>

```

```
</TableColumnItem>
<TableColumnItem>
    <PropertyName> Property for second column</PropertyName>
</TableColumnItem>
</TableColumnItems>
</TableRowEntry>
```

See Also

[Creating a Table View](#)

[EntrySelectedBy Element for TableRowEntry for TableControl](#)

[TableColumnItems Element for TableRowEntry for TableControl](#)

[TableRowEntries Element for TableControl](#)

[Wrap Element for TableRowEntry for TableControl](#)

[Writing a PowerShell Formatting File](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

EntrySelectedBy Element for TableRowEntry

Article • 09/17/2021

Defines the .NET types that use this definition of the table view or the condition that must exist for this definition to be used.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- EntrySelectedBy Element

Syntax

XML

```
<EntrySelectedBy>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <SelectionCondition>...</SelectionCondition>
</EntrySelectedBy>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `EntrySelectedBy` element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for TableControl	Optional element. Defines the condition that must exist for this table view definition to be used.
SelectionSetName Element for EntrySelectedBy for TableControl	Optional element. Specifies a set of .NET types that use this table view definition.
TypeName Element for EntrySelectedBy for TableControl	Optional element. Specifies a .NET type that uses this table view definition.

Parent Elements

[+] Expand table

Element	Description
TableRowEntry Element for TableControl	Defines the data that is displayed in a row of the table.

Remarks

You must specify at least one type, selection set, or selection condition for a table view definition. There is no maximum limit to the number of child elements that you can use.

Selection conditions are used to define a condition that must exist for the definition to be used, such as when an object has a specific property or that a specific property value or script evaluates to `true`. For more information about selection conditions, see [Defining Conditions for when a View Entry or Item is Used](#).

For more information about the components of a table view, see [Creating a Table View](#).

Example

The following example shows a `TableRowEntry` element that is used to display the properties of the `System.Diagnostics.Process` object.

XML

```
<TableRowEntry>
  <EntrySelectedBy>
    <TypeName>System.Diagnostics.Process</TypeName>
  </EntrySelectedBy>
  <TableColumnItems>
    < TableColumnItem>
      <PropertyName>PropertyForFirstColumn</PropertyName>
    </ TableColumnItem>
    < TableColumnItem>
      <PropertyName>PropertyForSecondColumn</PropertyName>
    </ TableColumnItem>
  </TableColumnItems>
</TableRowEntry>
```

See Also

[Creating a Table View](#)

[SelectionCondition Element for EntrySelectedBy for TableControl](#)

[SelectionSetName Element for EntrySelectedBy for TableControl](#)

[TableRowEntry Element for TableControl](#)

[TypeName Element for EntrySelectedBy for TableControl](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionCondition Element for EntrySelectedBy for TableControl

Article • 09/17/2021

Defines the condition that must exist to use for this definition of the table view. There is no limit to the number of selection conditions that can be specified for a table definition.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- EntrySelectedBy Element
- SelectionCondition Element

Syntax

XML

```
<SelectionCondition>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</SelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the SelectionCondition element.

Attributes

None.

Child Elements

[Expand table](#)

Element	Description
PropertyName Element for SelectionCondition for EntrySelectedBy for TableRowEntry	Optional element. Specifies the .NET property that triggers the condition.
ScriptBlock Element for SelectionCondition for EntrySelectedBy for TableRowEntry	Optional element. Specifies the script that triggers the condition.
SelectionSetName Element for SelectionCondition for EntrySelectedBy for TableRowEntry	Optional element. Specifies the set of .NET types that trigger the condition.
TypeName Element for SelectionCondition for EntrySelectedBy for TableRowEntry	Optional element. Specifies a .NET type that triggers the condition.

Parent Elements

[Expand table](#)

Element	Description
EntrySelectedBy Element for TableRowEntry	Defines the .NET types that use this table entry or the condition that must exist for this entry to be used.

Remarks

Each list entry must have at least one type name, selection set, or selection condition defined.

When you are defining a selection condition, the following requirements apply:

- The selection condition must specify a least one property name or a script block, but cannot specify both.

- The selection condition can specify any number of .NET types or selection sets, but cannot specify both.

For more information about how to use selection conditions, see [Defining Conditions for when a View Entry or Item is Used](#).

For more information about the components of a table view, see [Creating a Table View](#).

See Also

[Creating a Table View](#)

[Defining Conditions for When Data Is Displayed](#)

[EntrySelectedBy Element](#)

[PropertyName Element for SelectionCondition for EntrySelectedBy for TableRowEntry](#)

[ScriptBlock Element for SelectionCondition for EntrySelectedBy for TableRowEntry](#)

[SelectionSetName Element for SelectionCondition for EntrySelectedBy for TableRowEntry](#)

[TypeName Element for SelectionCondition for EntrySelectedBy for TableRowEntry](#)

[Writing a Windows PowerShell Formatting and Types File](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



[PowerShell feedback](#)

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PropertyName Element for SelectionCondition for EntrySelectedBy for TableRowEntry

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the table entry is used.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for TableRowEntry	Defines the condition that must exist for this table entry to be used.

Text Value

Specify the .NET property name.

Remarks

The selection condition must specify at least one property name or a script block, but cannot specify both. For more information about how selection conditions can be used, see [Defining Conditions for when a View Entry or Item is Used](#).

For more information about the components of a table view, see [Creating a Table View](#).

See Also

[Creating a Table View](#)

[Defining Conditions for When Data Is Displayed](#)

[ScriptBlock Element for SelectionCondition for EntrySelectedBy for TableRowEntry](#)

[SelectionCondition Element for EntrySelectedBy for TableRowEntry](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for SelectionCondition for EntrySelectedBy for TableControl

Article • 09/17/2021

Specifies the script block that triggers the condition. When this script is evaluated to `true`, the condition is met, and the table entry is used.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for TableRowEntry	Defines the condition that must exist for this table entry to be used.

Text Value

Specify the script that is evaluated.

Remarks

The selection condition must specify at least one script block or property name, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when a View Entry or Item is Used](#).

For more information about the components of a table view, see [Creating a Table View](#).

See Also

[Creating a Table View](#)

[Defining Conditions for When Data Is Displayed](#)

[PropertyName Element for SelectionCondition for EntrySelectedBy for TableRowEntry](#)

[SelectionCondition Element for EntrySelectedBy for TableRowEntry](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for SelectionCondition for EntrySelectedBy for TableControl

Article • 09/17/2021

Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met, and the object is displayed by using this definition of the table view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for TableRowEntry	Defines the condition that must exist to use for this definition of the table view.

Text Value

Specify the name of the selection set.

Remarks

The selection condition can specify a selection set or .NET type, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

Selection sets are common groups of .NET objects that can be used by any view that the formatting file defines. For more information about creating and referencing selection sets, see [Defining Sets of Objects](#).

For more information about other components of a wide view, see [Creating a Table View](#).

See Also

[Creating a Table View](#)

[Defining Conditions for When Data Is Displayed](#)

[TypeName Element for SelectionCondition for EntrySelectedBy for TableRowEntry](#)

[SelectionCondition Element for EntrySelectedBy for TableRowEntry](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for SelectionCondition for EntrySelectedBy for TableControl

Article • 09/17/2021

Specifies a .NET type that triggers the condition. When this type is present, the condition is met, and the table row is used.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof .NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for TableRowEntry	Defines the condition that must exist for this table row to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

The selection condition can specify any number of .NET types or selection sets, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when a View Entry or Item is Used](#).

For more information about the components of a table view, see [Creating a Table View](#).

See Also

[Creating a Table View](#)

[Defining Conditions for When Data Is Displayed](#)

[SelectionCondition Element for EntrySelectedBy for TableRowEntry](#)

[SelectionSetName Element for SelectionCondition for EntrySelectedBy for TableRowEntry](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

SelectionSetName Element for EntrySelectedBy for TableControl

Article • 09/17/2021

Specifies a set of .NET types the use this entry of the table view. There is no limit to the number of selection sets that can be specified for an entry.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- EntrySelectedBy Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent elements.

Attributes

None.

Child Elements

None.

Parent Elements

Element	Description
EntrySelectedBy Element	Defines the .NET types that use this entry or the condition that must exist for this entry to be used.

Text Value

Specify the name of the selection set.

Remarks

Selection sets are typically used when you want to define a group of objects that are used in multiple views. For example, you might want to create a table view and a list view for the same set of objects. For more information about defining selection sets, see [Defining Sets of objects for a View](#).

If you specify a selection set for an entry, you cannot specify a type name. For more information about how to specify a .NET type, see [TypeName Element for EntrySelectedBy for TableRowEntry](#).

For more information about the components of a table view, see [Creating a Table View](#).

See Also

[EntrySelectedBy Element](#)

[Defining Sets of objects for a View](#)

[Creating a Table View](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

TypeName Element for EntrySelectedBy for TableControl

Article • 09/17/2021

Specifies a .NET type that uses this entry of the table view. There is no limit to the number of types that can be specified for a table entry.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element
- TableRowEntry Element
- EntrySelectedBy Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof .NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
EntrySelectedBy Element	Defines the .NET types that use this entry or the condition that must exist for this entry to be used.

Text Value

Specify the name of the .NET type.

Remarks

Each list entry must have at least one type name, selection set, or selection condition defined.

For more information about the components of a table view, see [Creating a Table View](#).

See Also

[Creating a Table View](#)

[EntrySelectedBy Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Wrap Element for TableRowEntry

Article • 09/17/2021

Specifies that text that exceeds the column width is displayed on the next line. By default, text that exceeds the column width is truncated.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- TableControl Element
- TableRowEntries Element for TableControl
- TableRowEntry Element for TableRowEntries for TableControl
- Wrap Element for TableRowEntry for TableControl

Syntax

XML

```
<Wrap/>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent elements of the `Wrap` element.

Attributes

None.

Child Elements

None.

Parent Elements

Element	Description
TableRowEntry Element for TableRowEntries for TableControl	Defines the data that is displayed in a row of the table.

Remarks

For more information about the components of a table view, see [Creating a Table View](#).

See Also

[Creating a Table View](#)

[TableRowEntry Element for TableRowEntries for TableControl](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ViewSelectedBy Element

Article • 09/17/2021

Defines the .NET objects that are displayed by the view. Each view must specify at least one .NET object.

Schema

- ViewDefinitions Element
- View Element
- ViewSelectedBy Element

Syntax

XML

```
<ViewSelectedBy>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>SelectionSet</SelectionSetName>
</ViewSelectedBy>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `ViewSelectedBy` element. This element must contain at least one `TypeName` or `SelectionSetName` child element. There is no limit to the number of child elements that can be specified nor is their order significant.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
TypeName Element for ViewSelectedBy	Optional element. Specifies a .NET object that is displayed by the view.
SelectionSetName Element for ViewSelectedBy	Optional element. Specifies a set of .NET objects that are displayed by the view.

Parent Elements

[\[\] Expand table](#)

Element	Description
View Element	Defines a view that displays one or more .NET objects.

Remarks

For more information about how this element is used in different views, see [Table View Components](#), [List View Components](#), [Wide View Components](#), and [Custom Control Components](#).

The `SelectionSetName` element is used when the formatting file defines a set of objects that are displayed by multiple views. For more information about how selection sets are defined and referenced, see [Defining Sets of Objects](#).

Example

The following example shows how to specify the `System.ServiceProcess.ServiceController` object for a list view. The same schema is used for table, wide, and custom views.

XML

```
<View>
  <Name>System.ServiceProcess.ServiceController</Name>
  <ViewSelectedBy>
    <TypeName>System.ServiceProcess.ServiceController</TypeName>
  </ViewSelectedBy>
```

```
<ListControl>...</ListControl>  
</View>
```

See Also

[Creating a List View](#)

[Creating a Table View](#)

[Creating a Wide View](#)

[Creating Custom Controls](#)

[Defining Selection Sets](#)

[SelectionSetName Element for ViewSelectedBy](#)

[TypeName Element](#)

[Writing a PowerShell Formatting File](#)

SelectionSetName Element for ViewSelectedBy

Article • 09/17/2021

Specifies a set of .NET objects that are displayed by the view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ViewSelectedBy Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>Name of selection set<SelectionSetName>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
ViewSelectedBy Element	Defines the .NET objects that are displayed by the view.

Text Value

Specify the name of the selection set that is defined by the `Name` element for the selection set.

Remarks

You can use selection sets when you have a set of related objects that you want to reference by using a single name, such as a set of objects that are related through inheritance. For more information about defining and referencing selection sets, see [Defining Sets of Objects](#).

Example

The following example shows how to specify a selection set for a list view. The same schema is used for table, wide, and custom views.

XML

```
<View>
  <Name>Name of View</Name>
  <ViewSelectedBy>
    <SelectionSetName>NameofSelectionSet</SelectionSetName>
  </ViewSelectedBy>
  <ListControl>...</ListControl>
</View>
```

See Also

[Defining Selection Sets](#)

[ViewSelectedBy Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for ViewSelectedBy

Article • 09/17/2021

Specifies a .NET object that is displayed by the view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- ViewSelectedBy Element
- TypeName Element

Syntax

XML

```
<TypeName>FullyQualifiedTypeName</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent elements of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[+] Expand table

Element	Description
ViewSelectedBy Element	Defines the .NET objects that are displayed by the view.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

For more information about how this element is used in different views, see [Creating a Table View](#), [Creating a List View](#), [Creating a Wide View](#), and [Custom View Components](#).

Example

The following example shows how to specify the `System.ServiceProcess.ServiceController` object for a list view. The same schema is used for table, wide, and custom views.

```
XML
<View>
  <Name>System.ServiceProcess.ServiceController</Name>
  <ViewSelectedBy>
    <TypeName>System.ServiceProcess.ServiceController</TypeName>
  </ViewSelectedBy>
  <ListControl>...</ListControl>
</View>
```

See Also

[Creating a List View](#)

[Creating a Table View](#)

[Creating a Wide View](#)

[Creating Custom Controls](#)

[ViewSelectedBy Element](#)

[Writing a PowerShell Formatting File](#)

WideControl Element

Article • 09/17/2021

Defines a wide (single value) list format for the view. This view displays a single property value or script value for each object.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element

Syntax

XML

```
<WideControl>
  <AutoSize/>
  <ColumnNumber>PositiveInteger</ColumnNumber>
  <WideEntries>...</WideEntries>
</WideControl>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `WideControl` element. You cannot specify the `AutoSize` and `ColumnNumber` elements at the same time.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
AutoSize Element for WideControl	<p>Optional element.</p> <p>Specifies whether the column size and the number of columns are adjusted based on the size of the data.</p>
ColumnNumber Element for WideControl	<p>Optional element.</p> <p>Specifies the number of columns displayed in the wide view.</p>
WideEntries Element	<p>Required element.</p> <p>Provides the definitions of the wide view.</p>

Parent Elements

[\[+\] Expand table](#)

Element	Description
View Element	Defines a view that is used to display one or more .NET objects.

Remarks

When defining a wide view, you can add the `AutoSize` element or the `ColumnNumber` but you cannot add both.

In most cases, only one definition is required for each wide view, but it is possible to have multiple definitions if you want to use the same view to display different .NET objects. In those cases, you can provide a separate definition for each object or set of objects.

For more information about the components of a wide view, see [Wide View Components](#).

Example

The following example shows a `WideControl` element that is used to display a property of the `System.Diagnostics.Process` object.

XML

```
<View>
  <Name>process</Name>
  <ViewSelectedBy>
    <TypeName>System.Diagnostics.Process</TypeName>
  </ViewSelectedBy>
  <WideControl>
    <WideEntries>...</WideEntries>
  </WideControl>
</View>
```

For a complete example of a wide view, see [Wide View \(Basic\)](#).

See Also

[Autosize Element for WideControl](#)

[ColumnNumber Element for WideControl](#)

[View Element](#)

[WideEntries Element](#)

[Wide View \(Basic\)](#)

[Creating a Wide View](#)

[Writing a PowerShell Formatting File](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

AutoSize Element for WideControl

Article • 09/17/2021

Specifies whether the column size and the number of columns are adjusted based on the size of the data.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- Autosize Element

Syntax

XML

```
<AutoSize/>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `AutoSize` element.

Attributes

None.

Child Elements

None

Parent Elements

 Expand table

Element	Description
WideControl Element	Defines a wide (single value) list format for the view.

Remarks

When defining a wide view, you can add the `AutoSize` element or the `ColumnNumber` element, but you cannot add both.

For more information about the components of a wide view, see [Creating a Wide View](#).

For an example of a wide view, see [Wide View \(Basic\)](#).

See Also

[ColumnNumber Element for WideControl](#)

[Creating a Wide View](#)

[WideControl Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ColumnNumber Element for WideControl

Article • 09/17/2021

Specifies the number of columns displayed in the wide view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- ColumnNumber Element

Syntax

XML

```
<ColumnNumber>PositiveInteger</ColumnNumber>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ColumnNumber` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
WideControl Element	Defines a wide (single value) list format for the view.

Text Value

Specify a positive integer value.

Remarks

When defining a wide view, you can add the `AutoSize` element or the `ColumnNumber` element, but you cannot add both.

For more information about the components of a wide view, see [Creating a Wide View](#).

For an example of a wide view, see [Wide View \(Basic\)](#).

See Also

[Autosize Element for WideControl](#)

[Creating a Wide View](#)

[Wide View \(Basic\)](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

WideEntries Element

Article • 09/17/2021

Provides the definitions of the wide view. The wide view must specify one or more definitions.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element

Syntax

XML

```
<WideEntries>
  <WideEntry>...</WideEntry>
</WideEntries>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `WideEntries` element. At least one child element must be specified.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
WideEntry Element	Provides a definition of the wide view.

Parent Elements

[] [Expand table](#)

Element	Description
WideControl Element	Defines a wide (single value) list format for the view.

Remarks

A wide view is a list format that displays a single property value or script value for each object. For more information about the components of a wide view, see [Wide View Components](#).

Example

The following example shows a `WideEntries` element that defines a single `WideEntry` element. The `WideEntry` element contains a single `WideItem` element that defines what property or script value is displayed in the view.

XML

```
<WideControl>
  <WideEntries>
    <WideEntry>
      <WideItem>...</WideItem>
    <WideEntry>
  </WideEntries>
</WideControl>
```

For a complete example of a wide view, see [Wide View \(Basic\)](#).

See Also

[Creating a Wide View](#)

[WideControl Element](#)

[WideEntry Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

WideEntry Element

Article • 09/17/2021

Provides a definition of the wide view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element

Syntax

XML

```
<WideEntry>
  <EntrySelectedBy>...</EntrySelectedBy>
  <WideItem>...</WideItem>
</WideEntry>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `WideEntry` element. You must specify a single `WideItem` child element.

Attributes

None.

Child Elements

[+] Expand table

Element	Description
EntrySelectedBy Element for WideEntry	Optional element. Defines the .NET types that use this wide entry definition or the condition that must exist for this definition to be used.
WideItem Element	Required element. Defines the property or script whose value is displayed.

Parent Elements

[\[+\] Expand table](#)

Element	Description
WideEntries Element	Provides the definitions of the wide view.

Remarks

A wide view is a list format that displays a single property value or script value for each object. Unlike other types of views, you can specify only one item element for each view definition. For more information about the other components of a wide view, see [Creating a Wide View](#).

Example

The following example shows a `WideEntry` element that defines a single `WideItem` element. The `WideItem` element defines the property whose value is displayed in the view.

XML

```
<WideEntries>
  <WideEntry>
    <WideItem>
      <PropertyName>ProcessName</PropertyName>
    </WideItem>
  </WideEntry>
</WideEntries>
```

For a complete example of a wide view, see [Wide View \(Basic\)](#).

See Also

[Creating a Wide View](#)

[SelectionCondition Element for WideEntry](#)

[SelectionSetName Element for WideEntry](#)

[TypeName Element for WideEntry](#)

[WideEntries Element](#)

[WideItem Element](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

EntrySelectedBy Element for WideEntry

Article • 09/17/2021

Defines the .NET types that use this definition of the wide view or the condition that must exist for this definition to be used.

Schema

Configuration Element ViewDefinitions Element View Element WideControl Element
WideEntries Element WideEntry Element EntrySelectedBy Element

Syntax

XML

```
<EntrySelectedBy>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <SelectionCondition>...</SelectionCondition>
</EntrySelectedBy>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `EntrySelectedBy` element.

Attributes

None.

Child Elements

[] Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for WideEntry	Optional element. Defines the condition that must exist for this wide view definition to be used.

Element	Description
SelectionSetName Element for EntrySelectedBy for WideEntry	<p>Optional element.</p> <p>Specifies a set of .NET types that use this wide view definition.</p>
TypeName Element for EntrySelectedBy for WideEntry	<p>Optional element.</p> <p>Specifies a .NET type that uses this wide view definition.</p>

Parent Elements

[\[\] Expand table](#)

Element	Description
WideEntry Element	Provides a definition of the wide view.

Remarks

You must specify at least one type, selection set, or selection condition for a wide view definition. There is no maximum limit to the number of child elements that you can use.

Selection conditions are used to define a condition that must exist for the definition to be used, such as when an object has a specific property or that a specific property value or script value evaluates to `true`. For more information about selection conditions, see [Defining Conditions for Displaying Data](#).

For more information about other components of a wide view, see [Creating a Wide View](#).

See Also

[WideEntry Element](#)

[SelectionCondition Element for EntrySelectedBy for WideEntry](#)

[SelectionSetName Element for EntrySelectedBy for WideEntry](#)

[TypeName Element for EntrySelectedBy for WideEntry](#)

[Creating a Wide View](#)

Defining Conditions for Displaying Data

Writing a PowerShell Formatting File

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionCondition Element for EntrySelectedBy for WideControl

Article • 09/17/2021

Defines the condition that must exist for this definition to be used. There is no limit to the number of selection conditions that can be specified for a wide entry definition.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element
- EntrySelectedBy Element
- SelectionCondition Element

Syntax

XML

```
<SelectionCondition>
  <TypeName>Nameof.NetType</TypeName>
  <SelectionSetName>NameofSelectionSet</SelectionSetName>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToEvaluate</ScriptBlock>
</SelectionCondition>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionCondition` element. You must specify a single `PropertyName` or `ScriptBlock` element. The `SelectionSetName` and `TypeName` elements are optional. You can specify one of either element.

Attributes

None.

Child Elements

[Expand table](#)

Element	Description
PropertyName Element for SelectionCondition for EntrySelectedBy for WideEntry	Optional element. Specifies the .NET property that triggers the condition.
ScriptBlock Element for SelectionCondition for EntrySelectedBy for WideEntry	Optional element. Specifies the script block that triggers the condition.
SelectionSetName Element for SelectionCondition for EntrySelectedBy for WideEntry	Optional element. Specifies the set of .NET types that triggers the condition.
TypeName Element for SelectionCondition for EntrySelectedBy for WideEntry	Optional element. Specifies a .NET type that triggers the condition.

Parent Elements

[Expand table](#)

Element	Description
EntrySelectedBy Element for WideEntry	Defines the .NET types that use this wide entry or the condition that must exist for this entry to be used.

Remarks

Each wide entry must have at least one type name, selection set, or selection condition defined.

When you are defining a selection condition, the following requirements apply:

- The selection condition must specify at least one property name or a script block, but cannot specify both.
- The selection condition can specify any number of .NET types or selection sets, but cannot specify both.

For more information about how to use selection conditions, see [Defining Conditions for when a View Entry or Item is Used](#).

For more information about other components of a wide view, see [Creating a Wide View](#).

See Also

[Creating a Wide View](#)

[Defining Conditions for When Data Is Displayed](#)

[EntrySelectedBy Element for WideEntry](#)

[PropertyName Element for SelectionCondition for EntrySelectedBy for WideEntry](#)

[ScriptBlock Element for SelectionCondition for EntrySelectedBy for WideEntry](#)

[SelectionSetName Element for SelectionCondition for EntrySelectedBy for WideEntry](#)

[TypeName Element for SelectionCondition for EntrySelectedBy for WideEntry](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

PropertyName Element for SelectionCondition for EntrySelectedBy for WideEntry

Article • 09/17/2021

Specifies the .NET property that triggers the condition. When this property is present or when it evaluates to `true`, the condition is met, and the definition is used.

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

C#

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for WideEntry	Defines the condition that must exist for this definition to be used.

Text Value

Specify the .NET property name.

Remarks

The selection condition must specify at least one property name or a script to evaluate, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

For more information about other components of a wide view, see [Wide View](#).

See Also

[Creating a Wide View](#)

[Defining Conditions for When Data Is Displayed](#)

[ScriptBlock Element for SelectionCondition for EntrySelectedBy for WideEntry](#)

[SelectionCondition Element for EntrySelectedBy for WideEntry](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub



[PowerShell feedback](#)

PowerShell is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for SelectionCondition for EntrySelectedBy for WideControl

Article • 09/17/2021

Specifies the script that triggers the condition. When this script is evaluated to `true`, the condition is met, and the wide entry definition is used.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element
- EntrySelectedBy Element
- SelectionCondition
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToEvaluate</ScriptBlock>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for WideEntry	Defines the condition that must exist for this definition to be used.

Text Value

Specify the script that is evaluated.

Remarks

The selection condition must specify at least one script or property name to evaluate, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

For more information about other components of a wide view, see [Wide View](#).

See Also

[Creating a Wide View](#)

[Defining Conditions for When Data Is Displayed](#)

[PropertyName Element for SelectionCondition for EntrySelectedBy for WideEntry](#)

[SelectionCondition Element for EntrySelectedBy for WideEntry](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

SelectionSetName Element for SelectionCondition for EntrySelectedBy for WideEntry

Article • 09/17/2021

Specifies the set of .NET types that trigger the condition. When any of the types in this set are present, the condition is met, and the object is displayed by using this definition of the wide view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for WideEntry	Defines the condition that must exist for this definition to be used.

Text Value

Specify the name of the selection set.

Remarks

The selection condition can specify a selection set or .NET type, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

Selection sets are common groups of .NET objects that can be used by any view that the formatting file defines. For more information about creating and referencing selection sets, see [Defining Sets of Objects](#).

For more information about other components of a wide view, see [Creating a Wide View](#).

See Also

[Creating a Wide View](#)

[Defining Conditions for When Data Is Displayed](#)

[Defining Selection Sets](#)

[SelectionCondition Element for EntrySelectedBy for WideEntry](#)

[TypeName Element for SelectionCondition for EntrySelectedBy for WideEntry](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

TypeName Element for SelectionCondition for EntrySelectedBy for WideControl

Article • 09/17/2021

Specifies a .NET type that triggers the condition. When this type is present, the definition is used.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element
- EntrySelectedBy Element
- SelectionCondition Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof .NetType</TypeName>
```

Attributes and Elements

The following sections describe attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
SelectionCondition Element for EntrySelectedBy for WideEntry	Defines the condition that must exist for this wide entry to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

The selection condition can specify a .NET type or a selection set, but cannot specify both. For more information about how to use selection conditions, see [Defining Conditions for when Data is Displayed](#).

For more information about other components of a wide view, see [Creating a Wide View](#).

See Also

[Creating a Wide View](#)

[Defining Conditions for When Data Is Displayed](#)

[SelectionCondition Element for EntrySelectedBy for WideEntry](#)

[SelectionSetName Element for SelectionCondition for EntrySelectedBy for WideEntry](#)

[Writing a PowerShell Formatting File](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

TypeName Element for EntrySelectedBy for WideEntry

Article • 09/17/2021

Specifies a .NET type for the definition. The definition is used whenever this object is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element
- EntrySelectedBy Element
- TypeName Element

Syntax

XML

```
<TypeName>Nameof .NetType</TypeName>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `TypeName` element.

Attributes

None.

Child Elements

None.

Parent Elements

[] [Expand table](#)

Element	Description
EntrySelectedBy Element for WideEntry	Defines the .NET types that use this wide entry or the condition that must exist for this entry to be used.

Text Value

Specify the fully qualified name of the .NET type, such as `System.IO.DirectoryInfo`.

Remarks

Each wide entry must specify one or more .NET types, a selection set, or the selection condition that must exist for the definition to be used.

For more information about other components of a wide view, see [Creating a Wide View](#).

See Also

[Creating a Wide View](#)

[EntrySelectedBy Element for WideEntry](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SelectionSetName Element for EntrySelectedBy for WideControl

Article • 09/17/2021

Specifies a set of .NET objects for the definition. The definition is used whenever one of these objects is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element
- EntrySelectedBy Element
- SelectionSetName Element

Syntax

XML

```
<SelectionSetName>NameofSelectionSet</SelectionSetName>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `SelectionSetName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
EntrySelectedBy Element for WideEntry	Defines the .NET types that use this wide entry or the condition that must exist for this entry to be used.

Text Value

Specify the name of the selection set.

Remarks

Each definition must specify one type name, selection set, or selection condition.

Selection sets are typically used when you want to define a group of objects that are used in multiple views. For example, you might want to create a table view and a list view for the same set of objects. For more information about defining selection sets, see [Defining Sets of Objects for a View](#).

For more information about other components of a wide view, see [Creating a Wide View](#).

See Also

[Creating a Wide View](#)

[Defining Selection Sets](#)

[EntrySelectedBy Element for WideEntry](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

WideItem Element for WideControl

Article • 09/17/2021

Defines the property or script whose value is displayed.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element
- WideItem Element

Syntax

XML

```
<WideItem>
  <PropertyName>.NetTypeProperty</PropertyName>
  <ScriptBlock>ScriptToExecute</ScriptBlock>
  <FormatString>FormatPattern</FormatString>
</WideItem>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `WideItem` element. The `FormatString` element is optional. However, you must specify a `PropertyName` or `ScriptBlock` element, but you cannot specify both.

Attributes

None.

Child Elements

[] Expand table

Element	Description
FormatString Element for <code>WideItem</code> for <code>WideControl</code>	<p>Optional element.</p> <p>Specifies a format pattern that defines how the property or script value is displayed in the view.</p>
PropertyName Element for <code>WideItem</code>	<p>Specifies the property of the object whose value is displayed in the wide view.</p>
ScriptBlock Element for <code>WideItem</code>	<p>Specifies the script whose value is displayed in the wide view.</p>

Parent Elements

 [Expand table](#)

Element	Description
WideEntry Element	<p>Provides a definition of the wide view.</p>

Remarks

For more information about the components of a wide view, see [Wide View](#).

Example

The following example shows a `WideEntry` element that defines a single `WideItem` element. The `WideItem` element defines the property or script whose value is displayed in the view.

XML
<pre><WideEntry> <WideItem> <PropertyName>ProcessName</PropertyName> </WideItem> </WideEntry></pre>

For a complete example of a wide view, see [Wide View \(Basic\)](#).

See Also

[FormatString Element for Wideltem for WideControl](#)

[PropertyName Element for Wideltem](#)

[ScriptBlock Element for Wideltem](#)

[WideEntry Element](#)

[Writing a PowerShell Formatting File](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

FormatString Element for Wideltem

Article • 01/18/2022

Specifies a format pattern that defines how the property or script value is displayed in the view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element
- Wideltem Element
- FormatString Element

Syntax

XML

```
<FormatString>PropertyPattern</FormatString>
```

Attributes and Elements

The following sections describe the attributes, child elements, and the parent element of the `FormatString` element.

Attributes

None.

Child Elements

None.

Parent Elements

Element	Description
<code>WideItem</code> Element for <code>WideControl</code>	Defines the property or script whose value is displayed in a row of the list view.

Text Value

Specify the pattern that is used to format the data. For example, you can use this pattern to format the value of any property that is of type `System.TimeSpan`: {0:MMM}{0:dd}{0:HH}:{0:mm}.

Remarks

Format strings can be used when creating table views, list views, wide views, or custom views. For more information about formatting a value displayed in a view, see [Formatting Displayed Data](#).

For more information about using format strings in wide views, see [Creating a Wide View](#).

Example

The following example shows how to define a formatting string for the value of the `StartTime` property.

XML

```
<WideItem>
  <PropertyName>StartTime</PropertyName>
  <FormatString>{0:MMM} {0:DD} {0:HH}:{0:MM}</FormatString>
</WideItem>
```

See Also

[Creating a Wide View](#)

[WideItem Element for WideControl](#)

[Writing a Windows PowerShell Formatting and Types File](#)

PropertyName Element for Wideltem for WideControl

Article • 09/17/2021

Specifies the property of the object whose value is displayed in the wide view.

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element
- Wideltem Element
- PropertyName Element

Syntax

XML

```
<PropertyName>.NetTypeProperty</PropertyName>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `PropertyName` element.

Attributes

None.

Child Elements

None.

Parent Elements

 Expand table

Element	Description
WideItem Element	Defines the property or script whose value is displayed in the wide view.

Text Value

Specify the name of the property whose value is displayed.

Remarks

For more information about the components of a wide view, see [Creating a Wide View](#).

Example

This example shows a wide view that displays the value of the `ProcessName` property of the [System.Diagnostics.Process](#) object.

XML

```

View>
  <Name>process</Name>
  <ViewSelectedBy>
    <TypeName>System.Diagnostics.Process</TypeName>
  </ViewSelectedBy>
  <WideControl>
    <WideEntries>
      <WideEntry>
        <WideItem>
          <PropertyName>ProcessName</PropertyName>
        </WideItem>
      </WideEntry>
    </WideEntries>
  </WideControl>
</View>

```

See Also

[WideItem Element](#)

[Creating a Wide View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ScriptBlock Element for Wideltem for WideControl

Article • 09/17/2021

Specifies the script whose value is displayed in the wide view.

Schema

- Configuration Element
- ViewDefinitions Element
- View Element
- WideControl Element
- WideEntries Element
- WideEntry Element
- Wideltem Element
- ScriptBlock Element

Syntax

XML

```
<ScriptBlock>ScriptToExecute</ScriptBlock>
```

Attributes and Elements

The following sections describe the attributes, child elements, and parent element of the `ScriptBlock` element.

Attributes

None.

Child Elements

None.

Parent Elements

Element	Description
WideItem Element	Defines the property or script block whose value is displayed in the wide view.

Text Value

Specify the script whose value is displayed.

Remarks

For more information about the components of a wide view, see [Creating a Wide View](#).

Example

This example shows a `WideItem` element that defines a script whose value is displayed in the view.

XML

```
<WideItem>
  <ScriptBlock>ScriptToExecute</ScriptBlock>
</WideItem>
```

See Also

[WideItem Element](#)

[Creating a Wide View](#)

[Writing a PowerShell Formatting File](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review
issues and pull requests. For



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Writing Help for PowerShell Scripts and Functions

Article • 07/10/2023

PowerShell scripts and functions should be fully documented whenever they're shared with others. The `Get-Help` cmdlet displays the script and function help topics in the same format as it displays help for cmdlets, and all the `Get-Help` parameters work on script and function help topics.

PowerShell scripts can include a help topic about the script and help topics about each functions in the script. Functions that are shared independently of scripts can include their own help topics.

This document explains the format and correct placement of the help topics, and it suggests guidelines for the content.

Types of Script and Function Help

Comment-Based Help

The help topic that describes a script or function can be implemented as a set of comments within the script or function. When writing comment-based help for a script and for functions in a script, pay careful attention to the rules for placing the comment-based help. The placement determines whether the `Get-Help` cmdlet associates the help topic with the script or a function. For more information about writing comment-based help topics, see [about_Comment_Based_Help](#).

XML-Based Command Help

The help topic that describes a script or function can be implemented in an XML file that uses the command help schema. To associate the script or function with the XML file, use the `.EXTERNALHELP` comment keyword followed by the path and name of the XML file.

When the `.EXTERNALHELP` comment keyword is present, it takes precedence over comment-based help, even when `Get-Help` can't find a help file that matches the value of the `.EXTERNALHELP` keyword.

Online Help

You can post your help topics on the internet and then direct `Get-Help` to open the topics. For more information about writing comment-based help topics, see [Supporting Online Help](#).

There is no established method for writing conceptual ("About") topics for scripts and functions. However, you can post conceptual topics on the internet list the topics and their URLs in the Related Links section of a command help topic.

Content Considerations for Script and Function Help

- If you are writing a very brief help topic with only a few of the available command help sections, be sure to include clear descriptions of the script or function parameters. Also include one or two sample commands in the examples section, even if you decide to omit example descriptions.
- In all descriptions, refer to the command as a script or function. This information helps the user to understand and manage the command.

For example, the following detailed description states that the `New-Topic` command is a script. This reminds users that they need to specify the path and full name when they run it.

"The `New-Topic` script creates a blank conceptual topic for each topic name in the input file..."

The following detailed description states that `Disable-PSRemoting` is a function. This information is particularly useful to users when the session includes multiple commands with the same name, some of which might be hidden by a command with higher precedence.

The `Disable-PSRemoting` function disables all session configurations on the local computer...

- In a script help topic, explain how to use the script as a whole. If you are also writing help topics for functions in the script, mention the functions in your script help topic and include references to the function help topics in the Related Links section of the script help topic. Conversely, when a function is part of a script, explain in the function help topic the role that the function plays in the script and

how it might be used independently. Then list the script help topic in the Related Links section of the function help topic.

- When writing examples for a script help topic, be sure to include the path to the script file in the example command. This reminds users that they must specify the path explicitly, even when the script is in the current directory.
- In a function help topic, remind users that the function exists only in the current session and, to use it in other sessions, they need to add it, or add it a PowerShell profile.
- `Get-Help` displays the help topic for a script or function only when the script file and help topic files are saved in the correct locations. Therefore, it's not useful to include instructions for installing PowerShell, or saving or installing the script or function in a script or function help topic. Instead, include any installation instructions in the document that you use to distribute the script or function.

See Also

[Writing Comment-Based Help Topics](#)

Writing Comment-Based Help Topics

Article • 07/10/2023

You can write comment-based Help topics for functions and scripts using special Help comment keywords.

The `Get-Help` cmdlet displays comment-based Help in the same format in which it displays the cmdlet Help topics that are generated from XML files. Users can use all of the parameters of `Get-Help`, such as Detailed, Full, Example, and Online, to display function and script Help.

You can also write XML-based Help topics for scripts and functions and use the Help comment keywords to redirect users to the XML-based topics or other topics.

In This Section

- [Syntax of Comment-Based Help](#) - Describes the syntax of comment-based help.
- [Comment-Based Help Keywords](#) - Lists the keywords in comment-based help.
- [Placing Comment-Based Help in Functions](#) - Shows where to place comment-based help for a function.
- [Placing Comment-Based Help in Scripts](#) - Shows where to place comment-based help for a script.



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Syntax of Comment-Based Help

Article • 03/24/2025

This section describes the syntax of comment-based help.

Syntax Diagram

The syntax for comment-based Help is as follows:

```
# .< help keyword>
# <help content>
```

-or -

```
<#
.< help keyword>
< help content>
#>
```

Syntax Description

Comment-based Help is written as a series of comments. You can type a comment symbol (#) before each line of comments, or you can use the <# and #> symbols to create a comment block. All the lines within the comment block are interpreted as comments.

Each section of comment-based Help is defined by a keyword and each keyword is preceded by a dot (.). The keywords can appear in any order. The keyword names aren't case-sensitive.

A comment block must contain at least one help keyword. Some of the keywords, such as .EXAMPLE, can appear many times in the same comment block. The Help content for each keyword begins on the line after the keyword and can span multiple lines.

All the lines in a comment-based Help topic must be contiguous. If a comment-based Help topic follows a comment that isn't part of the Help topic, there must be at least

one blank line between the last non-Help comment line and the beginning of the comment-based Help.

For example, the following comment-based help topic contains the `.DESCRIPTION` keyword and its value, which is a description of a function or script.

PowerShell

```
<#
    .DESCRIPTION
        The Get-Function function displays the name and syntax of all functions
        in the session.
#>
```

Comment-Based Help Keywords

Article • 05/21/2025

This topic lists and describes the keywords in comment-based help.

Keywords in Comment-Based Help

The following are valid comment-based Help keywords. They're listed in the order in which they typically appear in a Help topic along with their intended use. These keywords can appear in any order in the comment-based Help, and they're not case-sensitive.

Note that the `.EXTERNALHELP` keyword takes precedence over all other comment-based help keywords. When `.EXTERNALHELP` is present, the [Get-Help](#) cmdlet doesn't display comment-based help, even when it can't find a help file that matches the value of the keyword.

.SYNOPSIS

A brief description of the function or script. This keyword can be used only once in each topic.

.DESCRIPTION

A detailed description of the function or script. This keyword can be used only once in each topic.

.PARAMETER <Parameter-Name>

The description of a parameter. You can include a `.PARAMETER` keyword for each parameter in the function or script.

The `.PARAMETER` keywords can appear in any order in the comment block, but the order in which the parameters appear in the `param` statement or function declaration determines the order in which the parameters appear in Help topic. To change the order of parameters in the Help topic, change the order of the parameters in the `param` statement or function declaration.

You can also specify a parameter description by placing a comment in the `param` statement immediately before the parameter variable name. If you use both a `param` statement comment and a `.PARAMETER` keyword, the description associated with the `.PARAMETER` keyword is used, and the `param` statement comment is ignored.

.EXAMPLE

A sample command that uses the function or script, optionally followed by sample output and a description. Repeat this keyword for each example.

.INPUTS

The .NET types of objects that can be piped to the function or script. You can also include a description of the input objects. Repeat this keyword for each input type.

.OUTPUTS

The .NET type of the objects that the cmdlet returns. You can also include a description of the returned objects. Repeat this keyword for each output type.

.NOTES

Additional information about the function or script.

.LINK

The name of a related topic. Repeat this keyword for each related topic. This content appears in the Related Links section of the Help topic.

The `.LINK` keyword content can also include a Uniform Resource Identifier (URI) to an online version of the same help topic. The online version opens when you use the **Online** parameter of `Get-Help`. The URI must begin with `http` or `https`.

.COMPONENT

The name of the technology or feature that the function or script uses, or to which it's related. The **Component** parameter of `Get-Help` uses this value to filter the search results returned by `Get-Help`.

.ROLE

The name of the user role for the help topic. The **Role** parameter of `Get-Help` uses this value to filter the search results returned by `Get-Help`.

.FUNCTIONALITY

The keywords that describe the intended use of the function. The **Functionality** parameter of `Get-Help` uses this value to filter the search results returned by `Get-Help`.

.FORWARDHELPTARGETNAME <Command-Name>

Redirects to the help topic for the specified command. You can redirect users to any help topic, including help content for a function, script, cmdlet, or provider.

PowerShell

```
# .FORWARDHELPTARGETNAME <Command-Name>
```

.FORWARDHELPCATEGORY

Specifies the help category of the item in `.FORWARDHELPTARGETNAME`. Valid values are `Alias`, `Cmdlet`, `HelpFile`, `Function`, `Provider`, `General`, `FAQ`, `Glossary`, `ScriptCommand`, `ExternalScript`, `Filter`, or `All`. Use this keyword to avoid conflicts when there are commands with the same name.

PowerShell

```
# .FORWARDHELPCATEGORY <Category>
```

.REMOTEHELPRUNSPACE <PSSession-variable>

Specifies a session that contains the help topic. Enter a variable that contains a `PSSession` object. This keyword is used by the `[Export-PSSession][09]` cmdlet to find the help content for the exported commands.

PowerShell

```
# .REMOTEHELPRUNSPACE <PSSession-variable>
```

.EXTERNALHELP

Specifies an XML-based help file for the script or function.

PowerShell

```
# .EXTERNALHELP <XML Help File>
```

The `.EXTERNALHELP` keyword is required when a function or script is documented in XML files. Without this keyword, `Get-Help` can't find the XML-based help file for the function or script.

The `.EXTERNALHELP` keyword takes precedence over other comment-based help keywords. If `.EXTERNALHELP` is present, `Get-Help` doesn't display comment-based help, even if it can't find a help topic that matches the value of the `.EXTERNALHELP` keyword.

If the function is exported by a module, set the value of the `.EXTERNALHELP` keyword to a filename without a path. `Get-Help` looks for the specified file name in a language-specific subdirectory of the module directory. There are no requirements for the name of the XML-based help file for a function, but a best practice is to use the following format:

Syntax

```
<ScriptModule.psm1>-help.xml
```

If the function isn't included in a module, include a path to the XML-based help file. If the value includes a path and the path contains UI-culture-specific subdirectories, `Get-Help` searches the subdirectories recursively for an XML file with the name of the script or function in accordance with the language fallback standards established for Windows, just as it does in a module directory.

For more information about the cmdlet help XML-based help file format, see [How to Write Cmdlet Help](#).

Placing Comment-Based Help in Functions

Article • 07/10/2023

This topic explains where to place comment-based help for a function so that the `Get-Help` cmdlet associates the comment-based help topic with the correct function.

Where to Place Comment-Based Help for a Function

- At the beginning of the function body.
- At the end of the function body.
- Before the `Function` keyword. When the function is in a script or script module, there can't be more than one blank line between the last line of the comment-based help and the `Function` keyword. Otherwise, `Get-Help` associates the help with the script, not with the function.

Examples of Help Placement in a Function

The following examples show each of the three placement options for comment-based help for a function.

Help at the Beginning of a Function Body

The following example shows comment-based at the beginning of a function body.

```
PowerShell

function MyProcess
{
    <#
        .DESCRIPTION
        The MyProcess function gets the Windows PowerShell process.
    #>

    Get-Process powershell
}
```

Help at the End of a Function Body

The following example shows comment-based help at the end of a function body.

PowerShell

```
function MyFunction
{
    Get-Process powershell

    <#
        .DESCRIPTION
        The MyProcess function gets the Windows PowerShell process.
    #>
}
```

Help Before the Function Keyword

The following examples shows comment-based help on the line before the function keyword.

PowerShell

```
<#
    .DESCRIPTION
    The MyProcess function gets the Windows PowerShell process.
#>
function MyFunction { Get-Process powershell}
```

Placing Comment-Based Help in Scripts

Article • 07/10/2023

This topic explains where to place comment-based help for a script so that the `Get-Help` cmdlet associates the comment-based help topic with scripts and not with any functions that might be in the script.

Where to Place Comment-Based Help for a Script

- At the beginning of the script file.

Script Help can be preceded in the script only by comments and blank lines.

- At the end of the script file.

If the first item in the script body (after the Help) is a function declaration, there must be at least two blank lines between the end of the script Help and the function declaration. Otherwise, the Help is interpreted as being Help for the function, not Help for the script.

Examples of Help Placement in a Script

The following examples show each of the placement options for comment-based help for a script.

Help at the Beginning of a Script

The following example shows comment-based at the beginning of a script.

```
PowerShell

<#
.DESCRIPTION
This script performs a series of network connection tests.
#>

param [string]$ComputerName
...
```

Help at the End of a Script

The following example shows comment-based at the end of a script.

PowerShell

```
...
function Ping { Test-Connection -ComputerName $ComputerName }

<#
.DESCRIPTION
This script performs a series of network connection tests.
#>
```

Autogenerated Elements of Comment-Based Help

Article • 03/24/2025

The `Get-Help` cmdlet automatically generates several elements of a comment-based topic. These autogenerated elements make comment-based help look very much like the help that's generated from XML files.

Autogenerated Elements

The `Get-Help` cmdlet automatically generates the following elements of a help topic. You can't edit these elements directly, but you can change the results by changing the source of the element.

Name

The Name section of a function Help topic is taken from the function name in the function definition. The Name of a script Help topic is taken from the script filename. To change the name or its capitalization, change the function definition or the script filename.

Syntax

The Syntax section of the Help topic is generated from the parameter list in the `param` statement of the function or script. To add detail to the Help topic syntax, such as the .NET type of a parameter, add the detail to the parameter list. If you don't specify a parameter type, the `Object` type is inserted as the default value.

Parameter List

The Parameters section of the Help topic is generated from the parameter list in the function or script and from the descriptions that you add using the `.PARAMETER` keyword or comments in the parameter list.

Parameters appear in the Parameters section in the same order that they appear in the parameter list. The spelling and capitalization of parameter names is also taken from the parameter list; it isn't affected by the parameter name specified by the `.PARAMETER` keyword.

Common Parameters

The common parameters are added to the syntax and parameter list of the Help topic, even if they have no effect. For more information about the common parameters, see [about_CommonParameters](#).

Parameter Attribute Table

`Get-Help` generates the table of parameter attributes that appears when you use the **Full** or **Parameter** parameter of `Get-Help`. The value of the **Required**, **Position**, and **Default** value attributes is taken from the function or script syntax.

Remarks

The Remarks section of the Help topic is automatically generated from the function or script name. You can't change or affect its content.

Examples of Comment-based Help

Article • 03/24/2025

This topic includes examples that demonstrate how to use comment-based help for scripts and functions.

Example 1: Comment-based Help for a Function

The following sample function includes comment-based Help.

PowerShell

```
function Add-Extension
{
    param ([string]$Name,[string]$Extension = "txt")
    $Name = $Name + "." + $Extension
    $Name

    <#
        .SYNOPSIS
        Adds a file name extension to a supplied name.

        .DESCRIPTION
        Adds a file name extension to a supplied name.
        Takes any strings for the file name or extension.

        .PARAMETER Name
        Specifies the file name.

        .PARAMETER Extension
        Specifies the extension. "Txt" is the default.

        .INPUTS
        None. You can't pipe objects to Add-Extension.

        .OUTPUTS
        System.String. Add-Extension returns a string with the extension or
        file name.

        .EXAMPLE
        PS> Add-Extension -Name "File"
        File.txt

        .EXAMPLE
        PS> Add-Extension -Name "File" -Extension "doc"
        File.doc

        .EXAMPLE
        PS> Add-Extension "File" "doc"
```

```
File.doc

.LINK
Online version: http://www.fabrikam.com/add-extension.html

.LINK
Set-Item
#>
}
```

The following output shows the results of a `Get-Help` command that displays the help for the `Add-Extension` function.

PowerShell

```
PS> Get-Help Add-Extension -Full
```

Output

NAME

Add-Extension

SYNOPSIS

Adds a file name extension to a supplied name.

SYNTAX

```
Add-Extension [[-Name] <String>] [[-Extension] <String>]
[<CommonParameters>]
```

DESCRIPTION

Adds a file name extension to a supplied name. Takes any strings for the file name or extension.

PARAMETERS

-Name

Specifies the file name.

Required? false

Position? 0

Default value

Accept pipeline input? false

Accept wildcard characters?

-Extension

Specifies the extension. "Txt" is the default.

Required? false

Position? 1

Default value

Accept pipeline input? false

Accept wildcard characters?

```
<CommonParameters>
    This cmdlet supports the common parameters: -Verbose, -Debug,
    -ErrorAction, -ErrorVariable, -WarningAction, -WarningVariable,
    -OutBuffer and -OutVariable. For more information, type
    "Get-Help about_CommonParameters".
```

INPUTS

None. You can't pipe objects to Add-Extension.

OUTPUTS

System.String. Add-Extension returns a string with the extension or file name.

----- EXAMPLE 1 -----

```
PS> Add-Extension -Name "File"
File.txt
```

----- EXAMPLE 2 -----

```
PS> Add-Extension -Name "File" -Extension "doc"
File.doc
```

----- EXAMPLE 3 -----

```
PS> Add-Extension "File" "doc"
File.doc
```

RELATED LINKS

Online version: <http://www.fabrikam.com/add-extension.html>
Set-Item

Example 2: Comment-based Help for a Script

The following sample function includes comment-based Help.

Notice the blank lines between the closing `#>` and the `param` statement. In a script that doesn't have a `param` statement, there must be at least two blank lines between the final comment in the Help topic and the first function declaration. Without these blank lines, `Get-Help` associates the Help topic with the function, instead of the script.

PowerShell

```
<#
.SYNOPSIS
Performs monthly data updates.

.DESCRIPTION
The Update-Month.ps1 script updates the registry with new data generated
```

```
 during the past month and generates a report.
```

```
.PARAMETER InputPath
```

```
Specifies the path to the CSV-based input file.
```

```
.PARAMETER OutputPath
```

```
Specifies the name and path for the CSV-based output file. By default,  
MonthlyUpdates.ps1 generates a name from the date and time it runs, and  
saves the output in the local directory.
```

```
.INPUTS
```

```
None. You can't pipe objects to Update-Month.ps1.
```

```
.OUTPUTS
```

```
None. Update-Month.ps1 doesn't generate any output.
```

```
.EXAMPLE
```

```
PS> .\Update-Month.ps1
```

```
.EXAMPLE
```

```
PS> .\Update-Month.ps1 -InputPath C:\Data\January.csv
```

```
.EXAMPLE
```

```
PS> .\Update-Month.ps1 -InputPath C:\Data\January.csv -OutputPath  
C:\Reports\2009\January.csv
```

```
#>
```

```
param ([string]$InputPath, [string]$OutputPath)
```

```
function Get-Data { }
```

The following command gets the script Help. Because the script isn't in a directory that's listed in the PATH environment variable, the `Get-Help` command that gets the script Help must specify the script path.

PowerShell

```
PS> Get-Help C:\ps-test\update-month.ps1 -Full
```

Output

NAME

```
C:\ps-test\Update-Month.ps1
```

SYNOPSIS

```
Performs monthly data updates.
```

SYNTAX

```
C:\ps-test\Update-Month.ps1 [-InputPath] <String> [[-OutputPath]  
<String>] [<CommonParameters>]
```

DESCRIPTION

The Update-Month.ps1 script updates the registry with new data generated during the past month and generates a report.

PARAMETERS

-InputPath

Specifies the path to the CSV-based input file.

Required?	true
Position?	0
Default value	
Accept pipeline input?	false
Accept wildcard characters?	

-OutputPath

Specifies the name and path for the CSV-based output file. By default, MonthlyUpdates.ps1 generates a name from the date and time it runs, and saves the output in the local directory.

Required?	false
Position?	1
Default value	
Accept pipeline input?	false
Accept wildcard characters?	

<CommonParameters>

This cmdlet supports the common parameters: -Verbose, -Debug, -ErrorAction, -ErrorVariable, -WarningAction, -WarningVariable, -OutBuffer and -OutVariable. For more information, type, "Get-Help about_CommonParameters".

INPUTS

None. You can't pipe objects to Update-Month.ps1.

OUTPUTS

None. Update-Month.ps1 doesn't generate any output.

----- EXAMPLE 1 -----

PS> .\Update-Month.ps1

----- EXAMPLE 2 -----

PS> .\Update-Month.ps1 -InputPath C:\Data\January.csv

----- EXAMPLE 3 -----

PS> .\Update-Month.ps1 -InputPath C:\Data\January.csv -OutputPath C:\Reports\2009\January.csv

RELATED LINKS

Example 3: Parameter Descriptions in a `param` Statement

This example shows how to insert parameter descriptions in the `param` statement of a function or script. This format is most useful when the parameter descriptions are brief.

PowerShell

```
function Add-Extension
{
    param
    (
        [string]
        # Specifies the file name.
        $Name,
        [string]
        # Specifies the file name extension. "Txt" is the default.
        $Extension = "txt"
    )
    $Name = $Name + "." + $Extension
    $Name

    <#
    .SYNOPSIS
    Adds a file name extension to a supplied name.
    ...
    #>
}
```

The results are the same as the results for Example 1. `Get-Help` interprets the parameter descriptions as though they were accompanied by the `.PARAMETER` keyword.

Example 4: Redirecting to an XML File

You can write XML-based Help topics for functions and scripts. Although comment-based Help is easier to implement, XML-based Help is required if you want more precise control over Help content or if you are translating Help topics into multiple languages. The following example shows the first few lines of the `Update-Month.ps1` script. The script uses the `.EXTERNALHELP` keyword to specify the path to an XML-based Help topic for the script.

PowerShell

```
# .EXTERNALHELP C:\MyScripts\Update-Month-Help.xml
```

```
param ([string]$InputPath, [string]$OutputPath)

function Get-Data { }
```

The following example shows the use of the `.EXTERNALHELP` keyword in a function.

```
PowerShell

function Add-Extension
{
    param ([string]$Name, [string]$Extension = "txt")
    $Name = $Name + "." + $Extension
    $Name

    # .EXTERNALHELP C:\ps-test\Add-Extension.xml
}
```

Example 5: Redirecting to a Different Help Topic

The following code is an excerpt from the beginning of the built-in `help` function in PowerShell, which displays one screen of Help text at a time. Because the Help topic for the `Get-Help` cmdlet describes the `Help` function, the `Help` function uses the `.FORWARDHELPTARGETNAME` and `.FORWARDHELPCATEGORY` keywords to redirect the user to the `Get-Help` cmdlet Help topic.

```
PowerShell

function help
{
    <#
        .FORWARDHELPTARGETNAME Get-Help
        .FORWARDHELPCATEGORY Cmdlet
    #>
    [CmdletBinding(DefaultParameterSetName='AllUsersView')]
    param(
        [Parameter(Position=0, ValueFromPipelineByPropertyName=$true)]
        [System.String]
        ${Name},
        ...
    }
```

The following command uses this feature. When a user types a `Get-Help` command for the `help` function, `Get-Help` displays the Help topic for the `Get-Help` cmdlet.

PowerShell

```
PS> Get-Help help
```

Output

NAME

Get-Help

SYNOPSIS

Displays information about Windows PowerShell cmdlets and concepts.

...

Writing Help for PowerShell Cmdlets

Article • 07/10/2023

PowerShell cmdlets can be useful, but unless your Help topics clearly explain what the cmdlet does and how to use it, the cmdlet may not get used or, even worse, it might frustrate users. The XML-based cmdlet Help file format enhances consistency, but great help requires much more.

If you have never written cmdlet Help, review the following guidelines. The XML schema required to author the cmdlet Help topic is described in the following section. Start with [Creating the Cmdlet Help File](#). That topic includes a description of the top-level XML nodes.

Writing Guidelines for Cmdlet Help

Write well

Nothing replaces a well-written topic. If you aren't a professional writer, find a writer or editor to help you. Another alternative is to copy your Help text into Microsoft Word and use the grammar and spelling checks to improve your work.

Write simply

Use simple words and phrases. Avoid jargon. Consider that many readers are equipped only with a foreign-language dictionary and your Help topic.

Write consistently

Help for related cmdlets should be similar (for example, `Get-Content` and `Set-Content`). Use the standard descriptions for standard parameters, like **Force** and **InputObject**. (Copy them from Help for the core cmdlets.) Use standard terms. For example, use "parameter", not "argument", and use "cmdlet" not "command" or "command-let."

Start the synopsis with a verb

The synopsis field informs the user what the cmdlet does, not what it's or how it works. Verbs create a task-based statement that informs users if this cmdlet meets their requirements. Use simple verbs like "get", "create", and "change." Avoid "set", which can be vague and fancy words like "modify".

Focus on objects

Most "get" cmdlets display something, but their primary function is to get an object. In your Help, focus on the object, so that users understand that the default display is one of many, and that they can use the methods and properties of the object that you retrieved for them in different ways.

Write detailed descriptions

Briefly list everything that the cmdlet can do in the detailed description. If the main function is to change one property, but the cmdlet can change all properties, list this in the detailed description.

Use conventional syntax

Use the standard Backus-Naur format which is common for Windows and Unix command-line Help.

Use Microsoft .NET types for parameter values

The placeholders for parameter values (in the syntax and parameter descriptions) show the .NET Framework types of the objects that the parameter will accept. The PowerShell team developed this convention to help teach users about the .NET Framework.

Write complete parameter descriptions

Parameter descriptions must inform users of two things: what the parameter does (its effect) and what they must type for the parameter values.

Write practical examples

The examples should show how to use all of the parameters, but the most important thing is to show how to use the cmdlet in real-world tasks. Start with a simple example and write increasingly complex examples. In the final example, show how to use the cmdlet in a pipeline.

Use the Notes field

Use the Notes field to explain concepts that users need to understand the cmdlet. You can also use notes to help users avoid common errors. Avoid URLs as they change.

Instead, provide users terms to search for.

Test your Help

Test the Help just like you test your code. Have friends and colleagues read your Help content and provide feedback. You can also solicit feedback from newsgroups.

See Also

- [How to Create the Cmdlet Help File](#)
- [How to Add the Cmdlet Name and Synopsis to a Cmdlet Help Topic](#)
- [How to Add the Detailed Description to a Cmdlet Help Topic](#)
- [How to Add Syntax to a Cmdlet Help Topic](#)
- [How to Add Parameters to a Cmdlet Help Topic](#)
- [How to add Input Types to a Cmdlet Help Topic](#)
- [How to Add Return Values to a Cmdlet Help Topic](#)
- [How to Add Notes to a Cmdlet Help Topic](#)
- [How to Add Examples to a Cmdlet Help Topic](#)
- [How to Add Related Links to a Cmdlet Help Topic](#)
- [Windows PowerShell SDK](#)

How to create the cmdlet help file

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This section describes how to create a valid XML file that contains content for Windows PowerShell cmdlet Help topics. This section discusses how to name the Help file, how to add the appropriate XML headers, and how to add nodes that will contain the different sections of the cmdlet Help content.

ⓘ Note

For a complete view of a Help file, open one of the `dll-Help.xml` files located in the Windows PowerShell installation directory. For example, the `Microsoft.PowerShell.Commands.Management.dll-Help.xml` file contains content for several of the PowerShell cmdlets.

How to create a cmdlet help file

1. Create a text file and save it using UTF8 encoding. The filename must have the following format so that Windows PowerShell can detect it as a cmdlet Help file.

```
<PSSnapInAssemblyName>.dll-Help.xml
```

2. Add the following XML headers to the text file. Be aware that the file will be validated against the Microsoft Assistance Markup Language (MAML) schema. Currently, PowerShell doesn't provide any tools to validate the file.

```
<?xml version="1.0" encoding="utf-8" ?> <helpItems xmlns="http://msh"  
schema="maml">
```

3. Add a **Command** node to the cmdlet Help file for each cmdlet in the assembly. Each node within the **Command** node relates to the different sections of the

cmdlet Help topic.

The following table lists the XML element for each node, followed by a descriptions of each node.

 Expand table

Node	Description
<details>	Adds content for the NAME and SYNOPSIS sections of the cmdlet Help topic. For more information, see How to Add the Cmdlet Name and Synopsis .
<maml:description>	Adds content for the DESCRIPTION section of the cmdlet Help topic. For more information, see How to Add the Detailed Description to a Cmdlet Help Topic .
<command:syntax>	Adds content for the SYNTAX section of the cmdlet Help topic. For more information, see How to Add Syntax to a Cmdlet Help Topic .
<command:parameters>	Adds content for the PARAMETERS section of the cmdlet Help topic. For more information, see How to Add Parameters to a Cmdlet Help Topic .
<command:inputTypes>	Adds content for the INPUTS section of the cmdlet Help topic. For more information, see How to Add Input Types to a Cmdlet Help Topic .
<command:returnValues>	Adds content for the OUTPUTS section of the cmdlet Help topic. For more information, see How to Add Return Values to a Cmdlet Help Topic .
<maml:alertset>	Adds content for the NOTES section of the cmdlet Help topic. For more information, see How to add Notes to a Cmdlet Help Topic .
<command:examples>	Adds content for the EXAMPLES section of the cmdlet Help topic. For more information, see How to Add Examples to a Cmdlet Help Topic .
<maml:relatedLinks>	Adds content for the RELATED LINKS section of the cmdlet Help topic. For more information, see How to Add Related Links to a Cmdlet Help Topic .

Example

Here is an example of a **Command** node that includes the nodes for the various sections of the cmdlet Help topic.

XML

```
<command:command
    xmlns:maml="http://schemas.microsoft.com/maml/2004/10"
    xmlns:command="http://schemas.microsoft.com/maml/dev/command/2004/10"
    xmlns:dev="http://schemas.microsoft.com/maml/dev/2004/10">
    <command:details>
        <!--Add name and synopsis here-->
    </command:details>
    <maml:description>
        <!--Add detailed description here-->
    </maml:description>
    <command:syntax>
        <!--Add syntax information here-->
    </command:syntax>
    <command:parameters>
        <!--Add parameter information here-->
    </command:parameters>
    <command:inputTypes>
        <!--Add input type information here-->
    </command:inputTypes>
    <command:returnValues>
        <!--Add return value information here-->
    </command:returnValues>
    <maml:alertSet>
        <!--Add Note information here-->
    </maml:alertSet>
    <command:examples>
        <!--Add cmdlet examples here-->
    </command:examples>
    <maml:relatedLinks>
        <!--Add links to related content here-->
    </maml:relatedLinks>
</command:command>
```

See also

- [How to Add the Cmdlet Name and Synopsis](#)
- [How to Add the Detailed Description to a Cmdlet Help Topic](#)
- [How to Add Syntax to a Cmdlet Help Topic](#)
- [How to Add Parameters to a Cmdlet Help Topic](#)
- [How to Add Input Types to a Cmdlet Help Topic](#)
- [How to Add Return Values to a Cmdlet Help Topic](#)
- [How to add Notes to a Cmdlet Help Topic](#)
- [How to Add Examples to a Cmdlet Help Topic](#)
- [How to Add Related Links to a Cmdlet Help Topic](#)
- [Windows PowerShell SDK](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to add the cmdlet name and synopsis to a cmdlet help topic

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This section describes how to add content that's displayed in the **NAME** and **SYNOPSIS** sections of the cmdlet help. In the Help file, this content is added to the Command node for each cmdlet.

ⓘ Note

For a complete view of a Help file, open one of the `dll-Help.xml` files located in the PowerShell installation directory. For example, the `Microsoft.PowerShell.Commands.Management.dll-Help.xml` file contains content for several of the PowerShell cmdlets.

To add the cmdlet name and a synopsis

- The cmdlet Help can display two descriptions for the cmdlet. The first description is a short description that's referred to as the synopsis. The second description is a more detailed description that's discussed in [Adding the Detailed Description to a Cmdlet Help Topic](#). Both these descriptions should be written as a single paragraph.
- In the synopsis don't repeat the cmdlet name. Informing the user that the `Get-Server` cmdlet gets a server is brief, but not informative. Instead, use synonyms and add details to the description.

Example: "Gets an object that represents a local or remote computer."

- Use simple verbs like "get", "create", and "change" in the synopsis. Avoid using "set" because it is vague, and fancy words such as "modify."

Example: "Gets information about the Authenticode signature in a file."

- Write in active voice. For example, "Use the TimeSpan object..." is much clearer than "the TimeSpan object can be used to..."
- Avoid the verb "display" when describing cmdlets that get objects. Although Windows PowerShell displays cmdlet data, it's important to introduce users to the concept that the cmdlet returns .NET Framework objects whose data may not be displayed. If you emphasize the display, the user might not realize that the cmdlet may have returned many other useful properties and methods that aren't displayed.

See also

[Windows PowerShell SDK](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to Add a Cmdlet Description

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This section describes how to add content that's displayed in the **DESCRIPTION** section of the cmdlet Help. In the Help file, this content is added to the **Command** node for each cmdlet.

ⓘ Note

For a complete view of a Help file, open one of the `dll-Help.xml` files located in the PowerShell installation directory. For example, the `Microsoft.PowerShell.Commands.Management.dll-Help.xml` file contains content for several of the PowerShell cmdlets.

To Add a Description

- Begin by explaining the basic features of the cmdlet in more detail. In many cases, you can explain the terms used in the cmdlet name and illustrate unfamiliar concepts with an example. For example, if the cmdlet appends data to a file, explain that it adds data to the end of an existing file.
- To find all of the features of the cmdlet, review the parameter list. Describe the primary function of the cmdlet, and then include other functions and features. For example, if the main function of the cmdlet is to change one property, but the cmdlet can change all of the properties, say so in the detailed description. If the cmdlet parameters let the users solicit information in different ways, explain it.
- Include information on ways that users can use the cmdlet, in addition to the obvious uses. For example, you can use the object that the `Get-Host` cmdlet retrieves to change the color of text in the Windows PowerShell command window.

Example: "The `Get-Acl` cmdlet gets objects that represent the security descriptor of a file or resource. The security descriptor contains the access control lists (ACLs) of the resource. The ACL specifies the permissions that users and user groups have to access the resource."

- The detailed description should describe the cmdlet, but it shouldn't describe concepts that the cmdlet uses. Place concept definitions in Additional Notes.

See Also

[Windows PowerShell SDK](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to add syntax to a cmdlet help topic

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

Before you start to code the XML for the syntax diagram in the cmdlet Help file, read this section to get a clear picture of the kind of data you need to provide, such as the parameter attributes, and how that data is displayed in the syntax diagram..

Parameter attributes

- Required
 - If true, the parameter must appear in all commands that use the parameter set.
 - If false, the parameter is optional in all commands that use the parameter set.
- Position
 - If named, the parameter name is required.
 - If positional, the parameter name is optional. When it's omitted, the parameter value must be in the specified position in the command. For example, if the value is position="1", the parameter value must be the first or only unnamed parameter value in the command.
- Pipeline Input
 - If true (ByValue), you can pipe input to the parameter. The input is associated with ("bound to") the parameter even if the property name and the object type don't match the expected type. The PowerShell parameter binding components try to convert the input to the correct type and fail the command only when the type can't be converted. Only one parameter in a parameter set can be associated by value.
 - If true (ByPropertyName), you can pipe input to the parameter. However, the input is associated with the parameter only when the parameter name matches the name of a property of the input object. For example, if the parameter name

is `Path`, objects piped to the cmdlet are associated with that parameter only when the object has a property named path.

- If true (ByValue, ByPropertyName), you can pipe input to the parameter either by property name or by value. Only one parameter in a parameter set can be associated by value.
- If false, you can't pipe input to this parameter.
- Globbing
 - If true, the text that the user types for the parameter value can include wildcard characters.
 - If false, the text that the user types for the parameter value can't include wildcard characters.

Parameter value attributes

- Required
 - If true, the specified value must be used whenever using the parameter in a command.
 - If false, the parameter value is optional. Typically, a value is optional only when it's one of several valid values for a parameter, such as in an enumerated type.

The **Required** attribute of a parameter value is different from the **Required** attribute of a parameter.

The required attribute of a parameter indicates whether the parameter (and its value) must be included when invoking the cmdlet. In contrast, the required attribute of a parameter value is used only when the parameter is included in the command. It indicates whether that particular value must be used with the parameter.

Typically, parameter values that are placeholders are required and parameter values that are literal aren't required, because they're one of several values that might be used with the parameter.

Gathering syntax information

1. Start with the cmdlet name.

SYNTAX
Get-Tech

2. List all the parameters of the cmdlet. Type a hyphen (-) before each parameter name. Separate the parameters into parameter sets (some cmdlets may have only one parameter set). In this example the Get-Tech cmdlet has two parameter sets.

SYNTAX

```
Get-Tech -Name -Type  
Get-Tech -Id -List -Type
```

Start each parameter set with the cmdlet name.

List the default parameter set first. The default parameter is specified by the cmdlet class.

For each parameter set, list its unique parameter first, unless there are positional parameters that must appear first. In the previous example, the Name and Id parameters are unique parameters for the two parameter sets (each parameter set must have one parameter that's unique to that parameter set). This makes it easier for users to identify what parameter they need to supply for the parameter set.

List the parameters in the order that they should appear in the command. If the order doesn't matter, list related parameters together, or list the most frequently used parameters first.

Be sure to list the WhatIf and Confirm parameters if the cmdlet supports ShouldProcess.

Don't list the common parameters (such as Verbose, Debug, and ErrorAction) in your syntax diagram. The `Get-Help` cmdlet adds that information for you when it displays the Help topic.

3. Add the parameter values. In PowerShell, parameter values are represented by their .NET type. However, the type name can be abbreviated, such as "string" for `System.String`.

SYNTAX

```
Get-Tech -Name string -Type Basic Advanced  
Get-Tech -Id int -List -Type Basic Advanced
```

Abbreviate types as long as their meaning is clear, such as `string` for `System.String` and `int` for `System.Int32`.

List all values of enumerations, such as the `-Type` parameter in the previous example, which can be set to **basic** or **advanced**.

Switch parameters, such as `-List` in the previous example, don't have values.

4. Add angle brackets to parameters values that are placeholder, as compared to parameter values that are literals.

SYNTAX

```
Get-Tech -Name <string> -Type Basic Advanced  
Get-Tech -Id <int> -List -Type Basic Advanced
```

5. Enclose optional parameters and their values in square brackets.

SYNTAX

```
Get-Tech -Name <string> [-Type Basic Advanced]  
Get-Tech -Id <int> [-List] [-Type Basic Advanced]
```

6. Enclose optional parameters names (for positional parameters) in square brackets.
The name for parameters that are positional, such as the `Name` parameter in the following example, don't have to be included in the command.

SYNTAX

```
Get-Tech [-Name] <string> [-Type Basic Advanced]  
Get-Tech -Id <int> [-List] [-Type Basic Advanced]
```

7. If a parameter value can contain multiple values, such as a list of names in the `Name` parameter, add a pair of square brackets directly following the parameter value.

SYNTAX

```
Get-Tech [-Name] <string[]> [-Type Basic Advanced]  
Get-Tech -Id <int[]> [-List] [-Type Basic Advanced]
```

8. If the user can choose from parameters or parameter values, such as the `Type` parameter, enclose the choices in curly brackets and separate them with the exclusive OR symbol();.

SYNTAX

```
Get-Tech [-Name] <string[]> [-Type {Basic | Advanced}]  
Get-Tech -Id <int[]> [-List] [-Type {Basic | Advanced}]
```

9. If the parameter value must use specific formatting, such as quotation marks or parentheses, show the format in the syntax.

SYNTAX

```
Get-Tech [-Name] <"string[]"> [-Type {Basic | Advanced}]  
Get-Tech -Id <int[]> [-List] [-Type {Basic | Advanced}]
```

Coding the syntax diagram XML

The syntax node of the XML begins immediately after the description node, which ends with the `</maml:description>` tag. For information about gathering the data used in the syntax diagram, see [Gathering Syntax Information](#).

Adding a syntax node

The syntax diagram displayed in the cmdlet Help topic is generated from the data in the syntax node of the XML. The syntax node is enclosed in a pair of `<command:syntax>` tags. With each parameter set of the cmdlet enclosed in a pair of `<command:syntaxitem>` tags. There is no limit to the number of `<command:syntaxitem>` tags that you can add.

The following example shows a syntax node that has syntax item nodes for two parameter sets.

XML

```
<command:syntax>  
  <command:syntaxItem>  
    ...  
    <!--Parameter Set 1 (default parameter set) parameters go here-->  
    ...  
  </command:syntaxItem>  
  <command:syntaxItem>  
    ...  
    <!--Parameter Set 2 parameters go here-->  
    ...
```

```
</command:syntaxItem>  
</command:syntax>
```

Adding the cmdlet name to the parameter set data

Each parameter set of the cmdlet is specified in a syntax item node. Each syntax item node begins with a pair of `<maml:name>` tags that include the name of the cmdlet.

The following example includes a syntax node that has syntax item nodes for two parameter sets.

XML

```
<command:syntax>  
  <command:syntaxItem>  
    <maml:name>Cmdlet-Name</maml:name>  
  </command:syntaxItem>  
  <command:syntaxItem>  
    <maml:name>Cmdlet-Name</maml:name>  
  </command:syntaxItem>  
</command:syntax>
```

Adding parameters

Each parameter added to the syntax item node is specified within a pair of `<command:parameter>` tags. You need a pair of `<command:parameter>` tags for each parameter included in the parameter set, with the exception of the common parameters that are provided by PowerShell.

The attributes of the opening `<command:parameter>` tag determine how the parameter appears in the syntax diagram. For information on parameter attributes, see [Parameter Attributes](#).

① Note

The `<command:parameter>` tag supports a child element `<maml:description>` whose content is never displayed. The parameter descriptions are specified in the parameter node of the XML. To avoid inconsistencies between the information in the syntax item nodes and the parameter node, omit the (`<maml:description>` or leave it empty.

The following example includes a syntax item node for a parameter set with two parameters.

XML

```
<command:syntaxItem>
  <maml:name>Cmdlet-Name</maml:name>
  <command:parameter required="true" globbing="true"
    pipelineInput="true (ByValue)" position="1">
    <maml:name>ParameterName1</maml:name>
    <command:parameterValue required="true">
      string[]
    </command:parameterValue>
  </command:parameter>
  <command:parameter required="true" globbing="true"
    pipelineInput="true (ByPropertyName)">
    <maml:name>ParameterName2</maml:name>
    <command:parameterValue required="true">
      int32[]
    </command:parameterValue>
  </command:parameter>
</command:syntaxItem>
```

How to add parameter information

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This section describes how to add content that's displayed in the **PARAMETERS** section of the cmdlet Help topic. The **PARAMETERS** section of the Help topic lists each of the parameters of the cmdlet and provides a detailed description of each parameter.

The content of the **PARAMETERS** section should be consistent with the content of the **SYNTAX** section of the Help topic. It's the responsibility of the Help author to make sure that both the **Syntax** and **Parameters** node contain similar XML elements.

ⓘ Note

For a complete view of a Help file, open one of the `dll-Help.xml` files located in the PowerShell installation directory. For example, the `Microsoft.PowerShell.Commands.Management.dll-Help.xml` file contains content for several of the PowerShell cmdlets.

To add parameters

1. Open the cmdlet Help file and locate the **Command** node for the cmdlet you are documenting. If you are adding a new cmdlet you will need to create a new **Command** node. Your Help file will contain a **Command** node for each cmdlet that you are providing Help content for. Here is an example of a blank **Command** node.

XML

```
<command:command>
</command:command>
```

2. Within the **Command** node, locate the **Description** node and add a **Parameters** node as shown below. Only one **Parameters** node is allowed, and it should immediately follow the **Syntax** node.

XML

```
<command:command>
  <command:details></command:details>
  <maml:description></maml:description>
  <command:syntax></command:syntax>
  <command:parameters>
    </command:parameters>
</command:command>
```

3. Within the **Parameters** node, add a **Parameter** node for each parameter of the cmdlet as shown below.

In this example, a **Parameter** node is added for three parameters.

XML

```
<command:parameters>
  <command:parameter></command:parameter>
  <command:parameter></command:parameter>
  <command:parameter></command:parameter>
</command:parameters>
```

Because these are the same XML tags that are used in the **Syntax** node, and because the parameters specified here must match the parameters specified by the **Syntax** node, you can copy the **Parameter** nodes from the **Syntax** node and paste them into the **Parameters** node. However, be sure to copy only one instance of a **Parameter** node, even if the parameter is specified in multiple parameter sets in the syntax.

4. For each **Parameter** node, set the attribute values that define the characteristics of each parameter. These attributes include the following: **required**, **globbing**, **pipelineInput**, and **position**.

XML

```
<command:parameters>
  <command:parameter required="true" globbing="true"
    pipelineInput="false" position="named">
  </command:parameter>
  <command:parameter required="false" globbing="false"
    pipelineInput="false" position="named">
  </command:parameter>
```

```
<command:parameter required="false" globbing="false"
    pipelineInput="false" position="named" ></command:parameter>
</command:parameters>
```

5. For each **Parameter** node, add the name of the parameter. Here is an example of the parameter name added to the **Parameter** node.

XML

```
<command:parameters>
    <command:parameter required="true" globbing="true"
        pipelineInput="false" position="named">
        <maml:name> Add parameter name... </maml:name>
    </command:parameter>
</command:parameters>
```

6. For each **Parameter** node, add the description of the parameter. Here is an example of the parameter description added to the **Parameter** node.

XML

```
<command:parameters>
    <command:parameter required="true" globbing="true"
        pipelineInput="false" position="named">
        <maml:name> Add parameter name... </maml:name>
        <maml:description>
            <maml:para> Add parameter description... </maml:para>
        </maml:description>
    </command:parameter>
</command:parameters>
```

7. For each **Parameter** node, add the .NET type of the parameter. The parameter type is displayed along with the parameter name.

Here is an example of the parameter .NET type added to the **Parameter** node.

XML

```
<command:parameters>
    <command:parameter required="true" globbing="true"
        pipelineInput="false" position="named">
        <maml:name> Add parameter name... </maml:name>
        <maml:description>
            <maml:para> Add parameter description... </maml:para>
        </maml:description>
        <dev:type> Add .NET Framework type... </dev:type>
    </command:parameter>
</command:parameters>
```

8. For each **Parameter** node, add the default value of the parameter. The following sentence is added to the parameter description when the content is displayed: **DefaultValue** is the default.

Here is an example of the parameter default value is added to the **Parameter** node.

XML

```
<command:parameters>
  <command:parameter required="true" globbing="true"
    pipelineInput="false" position="named">
    <maml:name> Add parameter name... </maml:name>
    <maml:description>
      <maml:para> Add parameter description... </maml:para>
    </maml:description>
    <dev:type> Add .NET Framework type... </dev:type>
    <dev:defaultValue> Add default value...</dev:defaultValue>
  </command:parameter>
</command:parameters>
```

9. For each Parameter that has multiple values, add a **possibleValues** node.

Here is an example of the of a **possibleValues** node that defines two possible values for the parameter

XML

```
<dev:possibleValues>
  <dev:possibleValue>
    <dev:value>Unknown</dev:value>
    <maml:description>
      <maml:para></maml:para>
    </maml:description>
  </dev:possibleValue>
  <dev:possibleValue>
    <dev:value>String</dev:value>
    <maml:description>
      <maml:para></maml:para>
    </maml:description>
  </dev:possibleValue>
</dev:possibleValues>
```

Here are some things to remember when adding parameters.

- The attributes of the parameter aren't displayed in all views of the cmdlet Help topic. However, they're displayed in a table following the parameter description when the user asks for the **Full** (`Get-Help <cmdletname> -Full`) or **Parameter** (`Get-Help <cmdletname> -Parameter`) view of the topic.

- The parameter description is one of the most important parts of a cmdlet Help topic. The description should be brief, as well as thorough. Also, remember that if the parameter description becomes too long, such as when two parameters interact with each other, you can add more content in the **NOTES** section of the cmdlet Help topic.

The parameter description provides two types of information.

- What the cmdlet does when the parameter is used.
- What a legal value is for the parameter.
- Because the parameter values are expressed as .NET objects, users need more information about these values than they would in a traditional command-line Help. Tell the user what type of data the parameter is designed to accept, and include examples.

The default value of the parameter is the value that's used if the parameter isn't specified on the command line. Note that the default value is optional, and isn't needed for some parameters, such as required parameters. However, you should specify a default value for most optional parameters.

The default value helps the user to understand the effect of not using the parameter. Describe the default value very specifically, such as the "Current directory" or the "PowerShell installation directory (`$PSHOME`)" for an optional path. You can also write a sentence that describes the default, such as the following sentence used for the **PassThru** parameter: "If **PassThru** isn't specified, the cmdlet doesn't pass objects down the pipeline." Also, because the value is displayed opposite the field name **Default value**, you don't need to include the term "default value" in the entry.

The default value of the parameter isn't displayed in all views of the cmdlet Help topic. However, it's displayed in a table (along with the parameter attributes) following the parameter description when the user asks for the **Full** (`Get-Help <cmdletname> -Full`) or **Parameter** (`Get-Help <cmdletname> -Parameter`) view of the topic.

The following XML shows a pair of `<dev:defaultValue>` tags added to the `<command:parameter>` node. Notice that the default value follows immediately after the closing `</command:parameterValue>` tag (when the parameter value is specified) or the closing `</maml:description>` tag of the parameter description. name.

XML

```
<command:parameters>
  <command:parameter required="true" globbing="true">
```

```

    pipelineInput="false" position="named">
<maml:name> Parameter name </maml:name>
<maml:description>
    <maml:para> Parameter Description </maml:para>
</maml:description>
<command:parameterValue required="true">
    Value
</command:parameterValue>
<dev:defaultValue> Default parameter value </dev:defaultValue>
</command:parameter>
</command:parameters>

```

Add Values for Enumerated Types

If the parameter has multiple values or values of an enumerated type, you can use an optional `<dev:possibleValues>` node. This node allows you to specify a name and description for multiple values.

Be aware that the descriptions of the enumerated values don't appear in any of the default Help views displayed by the `Get-Help` cmdlet, but other Help viewers may display this content in their views.

The following XML shows a `<dev:possibleValues>` node with two values specified.

XML

```

<command:parameters>
<command:parameter required="true" globbing="true"
    pipelineInput="false" position="named">
<maml:name> Parameter name </maml:name>
<maml:description>
    <maml:para> Parameter Description </maml:para>
</maml:description>
<command:parameterValue required="true">
    Value
</command:parameterValue>
<dev:defaultValue> Default parameter value </dev:defaultValue>
<dev:possibleValues>
    <dev:possibleValue>
        <dev:value> Value 1 </dev:value>
        <maml:description>
            <maml:para> Description 1 </maml:para>
        </maml:description>
    <dev:possibleValue>
        <dev:value> Value 2 </dev:value>
        <maml:description>
            <maml:para> Description 2 </maml:para>
        </maml:description>
    <dev:possibleValue>
</dev:possibleValues>

```

```
</command:parameter>  
</command:parameters>
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to add input types to a cmdlet help topic

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This section describes how to add an **INPUTS** section to a PowerShell cmdlet Help topic. The **INPUTS** section lists the .NET classes of objects that the cmdlet accepts as input from the pipeline, either by value or by property name.

There is no limit to the number of classes that you can add to an **INPUTS** section. The input types are enclosed in a `<command:inputTypes>` node, with each class enclosed in a `<command:inputType>` element.

The schema includes two `<maml:description>` elements in each `<command:inputType>` element. However, the `Get-Help` cmdlet displays only the content of the `<command:inputType>/<maml:description>` element.

Beginning in PowerShell 3.0, the `Get-Help` cmdlet displays the content of the `<maml:uri>` element. This element lets you direct users to topics that describe the .NET class.

The following XML shows the `<maml:inputTypes>` node.

XML

```
<command:inputTypes>
  <command:inputType>
    <dev:type>
      <maml:name> Class name </maml:name>
      <maml:uri> URI of a topic that describes the class </maml:uri>
      <maml:description/>
    </dev:type>
    <maml:description>
      <maml:para> Brief description </maml:para>
    </maml:description>
  </command:inputType>
</command:inputTypes>
```

```
</command:inputType>  
</command:inputTypes>
```

The following XML shows an example of using the `<maml:inputTypes>` node to document an input type.

XML

```
<command:inputTypes>  
  <command:inputType>  
    <dev:type>  
      <maml:name>System.DateTime</maml:name>  
  
      <maml:uri>https://learn.microsoft.com/dotnet/api/system.datetime</maml:uri>  
        <maml:description/>  
      </dev:type>  
      <maml:description>  
        <maml:para> You can pipe a date to the Set-Date cmdlet. <maml:para>  
        <maml:description>  
      </command:inputType>  
</command:inputTypes>
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to add return values to a cmdlet help topic

Article • 07/10/2023

This section describes how to add an **OUTPUTS** section to a PowerShell cmdlet Help topic. The **OUTPUTS** section lists the .NET classes of objects that the cmdlet returns or passes down the pipeline.

There is no limit to the number of classes that you can add to the **OUTPUTS** section. The return types of a cmdlet are enclosed in a `<command:returnValues>` node, with each class enclosed in a `<command:returnValue>` element.

If a cmdlet doesn't generate any output, use this section to indicate that there is no output. For example, in place of the class name, write **None** and provide a brief explanation. If the cmdlet generates output conditionally, use this node to explain the conditions and describe the conditional output.

The schema includes two `<maml:description>` elements in each `<command:returnValue>` element. However, the `Get-Help` cmdlet displays only the content of the `<command:returnValue>/<maml:description>` element.

Beginning in PowerShell 3.0, the `Get-Help` cmdlet displays the content of the `<maml:uri>` element. This element lets you direct users to topics that describe the .NET class.

The following XML shows the `<maml:returnValues>` node.

XML

```
<command:returnValues>
  <command:returnValue>
    <dev:type>
      <maml:name> Class Name </maml:name>
      <maml:uri> URI of a topic that describes the class </maml:uri>
      <maml:description/>
    </dev:type>
    <maml:description>
      <maml:para> Brief description <maml:para>
    </maml:description>
  </command:returnValue>
</command:returnValues>
```

The following XML shows an example of using the `<maml:returnValues>` node to document an output type.

XML

```
<command:returnValues>
  <command:returnValue>
    <dev:type>
      <maml:name> System.DateTime </maml:name>
      <maml:uri> https://learn.microsoft.com/dotnet/api/system.datetime
    </maml:uri>
    <maml:description/>
  </dev:type>
  <maml:description>
    <maml:para> Get-Date returns a DateTime object. <maml:para>
  </maml:description>
</command: returnValue>
</command: returnValues>
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to add notes to a cmdlet help topic

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This section describes how to add a **NOTES** section to a PowerShell cmdlet Help topic. The **NOTES** section is used to explain details that don't fit easily into the other structured sections, such as a more detailed explanation of a parameter. This content could include comments on how the cmdlet works with a specific provider, some unique, yet important, uses of the cmdlet, or ways to avoid possible error conditions.

The **NOTES** section is defined using a single `<maml:alertset>` node. There are no limits to the number of notes that you can add to a Notes section. For each note, add a pair of `<maml:alert>` tags to the `<maml:alertset>` node. The content of each note is added within a set of `<maml:para>` tags. Use blank `<maml:para>` tags for spacing.

XML

```
<maml:alertSet>
  <maml:title>Optional title for Note</maml:title>
  <maml:alert>
    <maml:para>Note 1</maml:para>
    <maml:para>Note a</maml:para>
  </maml:alert>
  <maml:alert>
    <maml:para>Note 2</maml:para>
  </maml:alert>
</maml:alertSet>
```

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to add examples to a cmdlet help topic

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

Things to know about examples in cmdlet help

- List all of the parameter names in the command, even when the parameter names are optional. This helps the user to interpret the command easily.
- Avoid aliases and partial parameter names, even though they work in PowerShell.
- In the example description, explain the rational for the construction of the command. Explain why you chose particular parameters and values, and how you use variables.
- If the command uses expressions, explain them in detail.
- If the command uses properties and methods of objects, especially properties that don't appear in the default display, use the example as an opportunity tell the user about the object.

Help Views that Display Examples

Examples appear only in the Detailed and Full views of cmdlet Help.

Adding an examples node

The following XML shows how to add an **Examples** node that contains a single **Example** node. Add additional example nodes for each examples you want to include in the topic.

XML

```
<command:examples>
  <command:example>
    </command:example>
</command:examples>
```

Adding an example title

The following XML shows how to add a **title** for the example. The **title** is used to set the example apart from other examples. PowerShell uses a standard header that includes a sequential example number.

XML

```
<command:examples>
  <command:example>
    <maml:title>----- EXAMPLE 1 -----</maml:title>
  </command:example>
</command:examples>
```

Adding preceding characters

The following XML shows how to add characters, such as the Windows PowerShell prompt, that are displayed immediately before the example command (without any intervening spaces). PowerShell uses the Windows PowerShell prompt: `C:\PS>`.

XML

```
<command:examples>
  <command:example>
    <maml:title>----- EXAMPLE 1 -----</maml:title>
    <maml:introduction>
      <maml:para>C:\PS></maml:para>
    </maml:introduction>
  </command:example>
</command:examples>
```

Adding the command

The following XML shows how to add the actual command of the example. When adding the command, type the entire name (do not use alias) of cmdlets and parameters. Also, use lowercase characters whenever possible.

XML

```
<command:examples>
  <command:example>
    <maml:title>----- EXAMPLE 1 -----</maml:title>
    <maml:introduction>
      <maml:para>C:\PS</maml:para>
    </maml:introduction>
    <dev:code> command </dev:code>
  </command:example>
</command:examples>
```

Adding a Description

The following XML shows how to add a description for the example. PowerShell uses a single set of `<maml:para>` tags for the description, even though multiple `<maml:para>` tags can be used.

XML

```
<command:examples>
  <command:example>
    <maml:title>----- EXAMPLE 1 -----</maml:title>
    <maml:introduction>
      <maml:para>C:\PS</maml:para>
    </maml:introduction>
    <dev:code> command </dev:code>
    <dev:remarks>
      <maml:para> command description </maml:para>
    </dev:remarks>
  </command:example>
</command:examples>
```

Adding example output

The following XML shows how to add the output of the command. The command results information is optional, but in some cases it's helpful to demonstrate the effect of using specific parameters. PowerShell uses two sets of blank `<maml:para>` tags to separate the command output from the command.

XML

```
<command:examples>
  <command:example>
    <maml:title>----- EXAMPLE 1 -----</maml:title>
    <maml:introduction>
```

```
<maml:para>C:\PS></maml:para>
</maml:introduction>
<dev:code> command </dev:code>
<dev:remarks>
  <maml:para> command description </maml:para>
  <maml:para></maml:para>
  <maml:para></maml:para>
  <maml:para> command output </maml:para>
</dev:remarks>
</command:example>
</command:examples>
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to add related links to a cmdlet help topic

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This section describes how to add references to other content that's related to a PowerShell cmdlet Help topic. Because these references appear in a command window, they don't link directly to the referenced content.

In the cmdlet Help topics that are included in PowerShell, these links reference other cmdlets, conceptual content (`about_`), and other documents and Help files that aren't related to PowerShell.

The following XML shows how to add a **RelatedLinks** node that contains two references to related topics.

XML

```
<maml:relatedLinks>
  <maml:navigationLink>
    <maml:linkText>Topic-name</maml:linkText>
  </maml:navigationLink>
  <maml:navigationLink>
    <maml:linkText>Topic-name</maml:linkText>
  </maml:navigationLink>
</ maml:relatedLinks >
```



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Writing Help for PowerShell Modules

Article • 07/10/2023

PowerShell modules can include Help topics about the module and about the module members, such as cmdlets, providers, functions and scripts. The `Get-Help` cmdlet displays the module Help topics in the same format as it displays Help for other PowerShell items, and users use standard `Get-Help` commands to get the Help topics.

This document explains the format and correct placement of module Help topics, and it suggests guidelines for module Help content.

Types of Module Help

A module can include the following types of Help.

- **Cmdlet Help.** The Help topics that describe cmdlets in a module are XML files that use the command help schema
- **Provider Help.** The Help topics that describe providers in a module are XML files that use the provider help schema.
- **Function Help.** The Help topics that describe functions in a module can be XML files that use the command help schema or comment-based Help topics within the function, or the script or script module
- **Script Help.** The Help topics that describe scripts in a module can be XML files that use the command help schema or comment-based Help topics in the script or script module.
- **Conceptual ("About") Help.** You can use a conceptual ("about") Help topic to describe the module and its members and to explain how the members can be used together to perform tasks. Conceptual Help topics are text files with Unicode (UTF-8) encoding. The filename must use the `about_<name>.help.txt` format, such as `about_MyModule.help.txt`. By default, PowerShell includes over 100 of these conceptual About Help topics, and they're formatted like the following example.

Output

TOPIC

`about_<subject or module name>`

SHORT DESCRIPTION

A short, one-line description of the topic contents.

LONG DESCRIPTION

A detailed, full description of the subject or purpose of the module.

EXAMPLES

Examples of how to use the module or how the subject feature works in practice.

KEYWORDS

Terms or titles on which you might expect your users to search for the information in this topic.

SEE ALSO

Text-only references for further reading. Hyperlinks can't work in the PowerShell console.

All the schema files can be found in the `$PSHOME\Scripts\PSMaml` folder.

Placement of Module Help

The `Get-Help` cmdlet looks for module Help topic files in language-specific subdirectories of the module directory.

For example, the following directory structure diagram shows the location of the Help topics for the `SampleModule` module.

```
<ModulePath>
  \SampleModule
    \<en-US>
      \about_SampleModule.help.txt
      \SampleModule.dll-help.xml
      \SampleNestedModule.dll-help.xml
    \<fr-FR>
      \about_SampleModule.help.txt
      \SampleModule.dll-help.xml
      \SampleNestedModule.dll-help.xml
```

ⓘ Note

In the example, the `<ModulePath>` placeholder represents one of the paths in the `PSModulePath` environment variable, such as `$HOME\Documents\Modules`, `$PSHOME\Modules`, or a custom path that the user specifies.

Getting Module Help

When a user imports a module into a session, the Help topics for that module are imported into the session along with the module. You can list the Help topic files in the value of the `FileList` key in the module manifest, but Help topics aren't affected by the `Export-ModuleMember` cmdlet.

You can provide module Help topics in different languages. The `Get-Help` cmdlet automatically displays module Help topics in the language that's specified for the current user in the Regional and Language Options item in Control Panel. In Windows Vista and later versions of Windows, `Get-Help` searches for the Help topics in language-specific subdirectories of the module directory in accordance with the language fallback standards established for Windows.

Beginning in PowerShell 3.0, running a `Get-Help` command for a cmdlet or function triggers automatic importing of the module. The `Get-Help` cmdlet immediately displays the contents of the help topics in the module.

If the module doesn't contain help topics and there are no help topics for the commands in the module on the user's computer, `Get-Help` displays auto-generated help. The auto-generated help includes the command syntax, parameters, and input and output types, but doesn't include any descriptions. The auto-generated help includes text that directs the user to try to use the `Update-Help` cmdlet to download help for the command from the internet or a file share. It also recommends using the `Online` parameter of the `Get-Help` cmdlet to get the online version of the help topic.

Supporting Updatable Help

Users of PowerShell 3.0 and later versions of PowerShell can download and install updated help files for a module from the internet or from a local file share. The `Update-Help` and `Save-Help` cmdlets hide the management details from the user. Users run the `Update-Help` cmdlet and then use the `Get-Help` cmdlet to read the newest help files for the module at the PowerShell command prompt. Users don't need to restart Windows or PowerShell.

Users behind firewalls and those without internet access can use Updatable Help, as well. Administrators with internet access use the `Save-Help` cmdlet to download and install the newest help files to a file share. Then, users use the `Path` parameter of the `Update-Help` cmdlet to get the newest help files from the file share.

Module authors can include help files in the module and use Updatable Help to update the help files, or omit help files from the module and use Updatable Help both to install and to update them.

For more information about Updatable Help, see [Supporting Updatable Help](#).

Supporting Online Help

Users who can't or don't install updated help files on their computers often rely on the online version of module help topics. The **Online** parameter of the `Get-Help` cmdlet opens the online version of a cmdlet or advanced function help topic for the user in their default internet browser.

The `Get-Help` cmdlet uses the value of the **HelpUri** property of the cmdlet or function to find the online version of the help topic.

Beginning in PowerShell 3.0, you can help users find the online version of cmdlet and function help topics by defining the **HelpUri** attribute on the cmdlet class or the **HelpUri** property of the **CmdletBinding** attribute. The value of the attribute is the value of the **HelpUri** property of the cmdlet or function.

For more information, see [Supporting Online Help](#).

See Also

- [Writing a PowerShell Module](#)
- [Supporting Updatable Help](#)
- [Supporting Online Help](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Naming Help files

Article • 03/24/2025

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This topic explains how to name an XML-based help file so that the [Get-Help](#) cmdlet can find it. The name requirements differ for each command type.

Cmdlet Help files

The help file for a C# cmdlet must be named for the assembly in which the cmdlet is defined. Use the following filename format:

```
<AssemblyName>.dll-help.xml
```

The assembly name format is required even when the assembly is a nested module.

For example, the [Get-WinEvent](#) cmdlet is defined in the `Microsoft.PowerShell.Diagnostics.dll` assembly. The [Get-Help](#) cmdlet looks for a help topic for the `Get-WinEvent` cmdlet only in the `Microsoft.PowerShell.Diagnostics.dll-help.xml` file in the module directory.

Provider Help files

The help file for a PowerShell provider must be named for the assembly in which the provider is defined. Use the following filename format:

```
<AssemblyName>.dll-help.xml
```

The assembly name format is required even when the assembly is a nested module.

For example, the Certificate provider is defined in the `Microsoft.PowerShell.Security.dll` assembly. The `Get-Help` cmdlet looks for a help topic for the Certificate provider only in the `Microsoft.PowerShell.Security.dll-help.xml` file in the module directory.

Function Help files

Functions can be documented using [comment-based help](#) or documented in an XML help file. When the function is documented in an XML file, the function must have an `.EXTERNALHELP` comment keyword that associates the function with the XML file.

Otherwise, the `Get-Help` cmdlet can't find the help file.

There are no technical requirements for the name of a function help file. However, a best practice is to name the help file for the script module in which the function is defined. For example, the following function is defined in the `MyModule.psm1` file.

```
C#  
  
#.EXTERNALHELP MyModule.psm1-help.xml  
function Test-Function { ... }
```

CIM Command Help files

The help file for a CIM command must be named for the CDXML file in which the CIM command is defined. Use the following filename format:

```
<FileName>.cdxml-help.xml
```

CIM commands are defined in CDXML files that can be included in modules as nested modules. When the CIM command is imported into the session as a function, PowerShell adds an `.EXTERNALHELP` comment keyword to the function definition that associates the function with an XML help file that is named for the CDXML file in which the CIM command is defined.

Script Workflow Help files

Script workflows that are included in modules can be documented in XML-based help files. There are no technical requirements for the name of the help file. However, a best practice is to name the help file for the script module in which the script workflow is defined. For example:

<ScriptModule>.psm1-help.xml

Unlike other scripted commands, script workflows don't require an `.EXTERNALHELP` comment keyword to associate them with a help file. Instead, PowerShell searches the UI-Culture-specific subdirectories of the module directory for XML-based help files and looks for help for the script workflow in all the files. `.EXTERNALHELP` comment keyword are ignored.

Because the `.EXTERNALHELP` comment keyword is ignored, the `Get-Help` cmdlet can find help for script workflows only when they're included in modules.

Supporting Updatable Help

Article • 07/10/2023

The *Windows PowerShell Updatable Help System*, introduced in Windows PowerShell 3.0, is designed to ensure that users always have the newest help topics at the command prompt on their local computer. Along with Windows PowerShell online help, Updatable Help provides a complete help solution for users. This section describes the Updatable Help System and explains how module authors can support Updatable Help for their modules.

This section includes the following topics.

- [Updatable Help Overview](#)
- [Updatable Help Authoring: Step-by-Step](#)
- [How Updatable Help Works](#)
- [How to Create a HelpInfo XML File](#)
- [How to Prepare Updatable Help CAB Files](#)
- [How to Update Help Files](#)
- [How to Test Updatable Help](#)

See Also

- [Supporting Online Help](#)
- [Updatable Help Status Table](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Updatable Help Overview

Article • 07/10/2023

This document provides a basic introduction to the design and operation of the PowerShell Updatable Help feature. It's designed for module authors and others who deliver Windows PowerShell help topics to users.

Introduction

PowerShell help topics are an integral part of the PowerShell experience. Like PowerShell modules, help topics are continually updated and improved by the authors and by the contributions of the community of PowerShell users.

The *Updatable Help* feature, introduced in Windows PowerShell 3.0, ensures that users have the newest versions of help topics at the command prompt, even for built-in PowerShell commands, without downloading new modules or running Windows Update. Updatable Help makes updating simple by providing cmdlets that download the newest versions of help topics from the internet and install them in the correct subdirectories on the user's local computer. Even users who are behind firewalls can use the new cmdlets to get updated help from an internal file share.

Updatable Help is fully supported by all Windows PowerShell modules in Windows 8 and Windows Server 2012, and its features are available to all Windows PowerShell module authors. Updatable Help supports only XML-based help files. It doesn't support comment-based help.

Updatable Help includes the following features.

- The [Update-Help](#) cmdlet, which determines whether users have the newest help files for a module and, if not, downloads the newest help files from the internet, unpacks them, and installs them in the correct module subdirectories on the user's computer. Users can use the [Get-Help](#) cmdlet to view the newly-installed help topics immediately. They don't need to restart PowerShell.
- The [Save-Help](#) cmdlet, which downloads the newest help files from the internet and saves them in a file system directory. Users can use the [Update-Help](#) cmdlet to get help files from the file system directory, and unpack and install them in the module subdirectories on the user's computer. The [Save-Help](#) cmdlet is designed for users who have limited or no internet access and for enterprises who prefer to limit internet access.

- **Help for a Module.** Help files for a module are managed and delivered as a unit, so users can get all of the help files for the modules they use. Updatable help is supported only for modules, not for Windows PowerShell snap-ins.
- **Version support.** Updatable Help uses standard four-position (N1.N2.N3.N4) version numbers. Updatable Help downloads help files when the version number of the help files on the user's computer (or in the `Save-Help` directory) is lower than the version number of the help files at the internet location.
- **Multi-language support.** Updatable Help supports module help files in multiple UI cultures. Updatable Help filenames include standard language codes, such as "en-US" and "ja-JP", and the `Update-Help` and `Save-Help` cmdlets place the help files into language-specific subdirectories of the module directory.
- **Auto-generated help.** The `Get-Help` cmdlet displays basic help for commands that don't have help files. The auto-generated help includes the command syntax and aliases, and instructions for using online help and Updatable Help.
- **Enhanced Online help.** Easy access to online help no longer requires help files. The `Online` parameter of the `Get-Help` cmdlet now gets the URL of an online help topic from the value of the `HelpUri` property of any command, if it can't find the online help URL in a help file. You can populate the `HelpUri` property by adding a `HelpUri` attribute to the code of cmdlets, functions, and CIM commands, or using the `.LINK` comment-based help keyword in workflows and scripts.

To make our help files updatable, the Windows PowerShell modules in Windows don't come with help files. Users can use Updatable Help to install help files and update them. Authors of other modules can include help files in modules or omit them. Support for Updatable Help is optional, but recommended.

Updatable Help Authoring: Step-by-Step

Article • 07/10/2023

This article documents the steps required to publish Updatable Help.

Authoring Updatable Help: Step-by-Step

Updatable Help is designed for end-users, but it also provides significant benefits to module authors and help writers, including the ability to add content, fix errors, deliver in multiple UI cultures, and respond to user comments and requests, long after the module has shipped. This topic explains how you package and upload help files so that users can download and install them using the [Update-Help](#) and [Save-Help](#) cmdlets.

The following steps provide an overview of the process of supporting Updatable Help.

Step 1: Find an internet site for your help files

The first step in creating updatable help is to find an internet location for your module's help files. Actually, you can use two different locations. You can keep your module's help information file (HelpInfo XML - described below) at one internet location and the help content files (CAB and ZIP) at another internet location. All help content files for a module must be in the same location. You can place help content files for different modules in the same location.

Step 2: Add a HelpInfoURI key to your module manifest

Add a **HelpInfoURI** key to your module manifest. The value of the key is the Uniform Resource Identifier (URI) of the location of the HelpInfo XML information file for your module. For security, the address must begin with `http:` or `https:`. The URI should specify an internet location for the HelpInfo XML file. Don't include the HelpInfo XML filename.

For example:

PowerShell

```
@{
    RootModule = TestModule.psm1
    ModuleVersion = '2.0'
```

```
    HelpInfoURI = 'https://go.microsoft.com/fwlink/?LinkID=0123'  
}
```

ⓘ Note

The **HelpInfoURI** must end with a forward slash (/) character or redirect to a location that ends with a forward slash (/).

Step 3: Create a HelpInfo XML file

The HelpInfo XML information file contains the URI of the internet location of your help files and the version numbers of the newest help files for your module in each supported UI culture. Every PowerShell module has one HelpInfo XML file. When you update your help files, you must update the HelpInfo XML file. For more information, see [How to Create a HelpInfo XML File](#).

Step 4: Create CAB and ZIP files

PowerShell on Windows expects the help content files a module to be stored in a CAB file. PowerShell on Linux or macOS expects the help content files a module to be stored in a ZIP file. If your module runs across multiple platforms you must create both formats.

Use a tool, such as `MakeCab.exe`, to create a CAB file that contains the help files for your module. Create a separate CAB file for the help files in each supported UI culture. For more information, see [How to Prepare Updatable Help CAB Files](#).

You can use the [Compress-Archive cmdlet](#) to create a ZIP file.

Step 5: Upload your files

To publish new or updated help files, upload the help content files to the internet location specified by the **HelpContentUri** element in the HelpInfo XML file. Then, upload the HelpInfo XML file to the internet location specified by the value of the **HelpInfoUri** key in the module manifest.

Using PlatyPS to create help content

PlatyPS is a PowerShell module designed to help you create Help content for your modules. You author the help content in Markdown files. PlatyPS can create Markdown

templates for your cmdlet, convert the Markdown files to the XML help format (MAML), create HelpInfo XML files, and package the MAML help content into CAB and ZIP files.

For more information, see [Create XML-based help using PlatyPS](#).

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How Updatable Help Works

Article • 07/10/2023

This topic explains how Updatable Help processes the HelpInfo XML file and CAB files for each module, and installs updated help for users.

The Update-Help Process

The following list describes the actions of the [Update-Help](#) cmdlet when a user runs a command to update the help files for a module in a particular UI culture.

1. `Update-Help` gets the remote HelpInfo XML file from the location specified by the value of the **HelpInfoURI** key in the module manifest and validates the file against the schema. (To view the schema, see [HelpInfo XML Schema](#).) Then `Update-Help` looks for a local HelpInfo XML file for the module in the module directory on the user's computer.
2. `Update-Help` compares the version number of the help files for the specified UI culture in the remote and local HelpInfo XML files for the module. If the version number on the remote file is greater than version number on the local file, or if there is no local HelpInfo XML file for the module, `Update-Help` prepares to download new help files.
3. `Update-Help` selects the CAB file for the module from the location specified by the **HelpContentUri** element in the remote HelpInfo XML file. It uses the module name, module GUID, and UI culture to identify the CAB file.
4. `Update-Help` downloads the CAB file, unpacks it, validates the help content files, and saves the help content files in the language-specific subdirectory of the module directory on the user's computer.
5. `Update-Help` creates a local HelpInfo XML file by copying the remote HelpInfo XML file. It edits the local HelpInfo XML file so that it includes elements only for the CAB file that it installed. Then it saves the local HelpInfo XML file in the module directory and concludes the update.

The Save-Help Process

The following list describes the actions of the [Save-Help](#) and [Update-Help](#) cmdlets when a user runs commands to update the help files in a file share, and then use those files to

update the help files on the user's computer.

The `Save-Help` cmdlet performs the following actions in response to a command to save the help files for a module in a file share that's specified by the **DestinationPath** parameter.

1. `Save-Help` gets the remote HelpInfo XML file from the location specified by the value of the **HelpInfoURI** key in the module manifest and validates the file against the schema. (To view the schema, see [HelpInfo XML Schema](#).) Then `Save-Help` looks for a local HelpInfo XML file in the directory that's specified by the **DestinationPath** parameter in the `Save-Help` command.
2. `Save-Help` compares the version number of the help files for the specified UI culture in the remote and local HelpInfo XML files for the module. If the version number on the remote file is greater than version number on the local file, or if there is no local HelpInfo XML file for the module in the **DestinationPath** directory, `Save-Help` prepares to download new help files.
3. `Save-Help` selects the CAB file for the module from the location specified by the **HelpContentUri** element in the remote HelpInfo XML file. It uses the module name, module GUID, and UI culture to identify the CAB file.
4. `Save-Help` downloads the CAB file and saves it in the **DestinationPath** directory. (It does not create any language-specific subdirectories.)
5. `Save-Help` creates a local HelpInfo XML file by copying the remote HelpInfo XML file. It edits the local HelpInfo XML file so that it includes elements only for the CAB file that it saved. Then it saves the local HelpInfo XML file in the **DestinationPath** directory and concludes the update.

The `Update-Help` cmdlet performs the following actions in response to a command to update the help files on a user's computer from the files in a file share that's specified by the **SourcePath** parameter.

6. `Update-Help` gets the remote HelpInfo XML file from the **SourcePath** directory. Then it looks for a local HelpInfo XML file in the module directory on the user's computer.
7. `Update-Help` compares the version number of the help files for the specified UI culture in the remote and local HelpInfo XML files for the module. If the version number on the remote file is greater than version number on the local file, or if there is no local HelpInfo XML file, `Update-Help` prepares to install new help files.

8. `Update-Help` selects the CAB file for the module from **SourcePath** directory. It uses the module name, module GUID, and UI culture to identify the CAB file.
9. `Update-Help` unpacks the CAB file, validates the help content files, and saves the help content files in the language-specific subdirectory of the module directory on the user's computer.
10. `Update-Help` creates a local HelpInfo XML file by copying the remote HelpInfo XML file. It edits the local HelpInfo XML file so that it includes elements only for the CAB file that it installed. Then it saves the local HelpInfo XML file in the module directory and concludes the update.

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to create a HelpInfo XML file

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This topics in this section explains how to create and populate a help information file, commonly known as a "HelpInfo XML file," for the PowerShell Updatable Help feature.

HelpInfo XML file overview

The HelpInfo XML file is the primary source of information about Updatable Help for the module. It includes the location of the help files for the modules, the supported UI cultures, and the version numbers that Updatable Help uses to determine whether the user has the newest help files.

Each module has just one HelpInfo XML file, even if the module includes multiple help files for multiple UI cultures. The module author creates the HelpInfo XML file and places it in the internet location that's specified by the **HelpInfoUri** key in the module manifest. When the module help files are updated and uploaded, the module author updates the HelpInfo XML file and replaces the original HelpInfo XML file with the new version.

It's critical that the HelpInfo XML file is carefully maintained. If you upload new files, but forget to increment the version numbers, Updatable Help will not download the new files to users' computers. If you add help files for a new UI culture, but don't update the HelpInfo XML file or place it in the correct location, Updatable Help will not download the new files.

In this section

This section includes the following topics.

- [HelpInfo XML Schema](#)

- [HelpInfo XML Sample File](#)
- [How to Name a HelpInfo XML File](#)
- [How to Set HelpInfo XML Version Numbers](#)

See also

[Supporting Updatable Help](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

HelpInfo XML Schema

Article • 07/10/2023

This topic contains the XML schema for Updatable Help Information files, commonly known as "HelpInfo XML files."

HelpInfo XML Schema

HelpInfo XML files are based on the following XML schema.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<schema elementFormDefault="qualified"
targetNamespace="http://schemas.microsoft.com/powershell/help/2010/05"
xmlns="http://www.w3.org/2001/XMLSchema">
<element name="HelpInfo">
<complexType>
<sequence>
<element name="HelpContentURI" type="anyURI" minOccurs="1"
maxOccurs="1" />
<element name="SupportedUICultures" minOccurs="1" maxOccurs="1">
<complexType>
<sequence>
<element name="UICulture" minOccurs="1" maxOccurs="unbounded">
<complexType>
<sequence>
<element name="UICultureName" type="language"
minOccurs="1" maxOccurs="1" />
<element name="UICultureVersion" type="string"
minOccurs="1" maxOccurs="1" />
</sequence>
</complexType>
</element>
</sequence>
</complexType>
</element>
</sequence>
</complexType>
</element>
</schema>
```

HelpInfo XML Elements

The HelpInfo XML file includes the following elements.

- **HelpContentURI** - Contains the URI of the location of the help CAB files for the module. The URI must begin with "http" or "https". The URI should specify an internet location, but must not include the CAB filename. The **HelpContentURI** value can be the same or different from the **HelpInfoURI** value.
- **SupportedUICultures** - Represents the module help files in all UI cultures. Contains **UICulture** elements, each of which represents a set of help files for the module in a specified UI culture.
- **UICulture** - Represents a set of help files for the module in a specified UI culture. Add a **UICulture** element for each UI culture in which the help files are written.
- **UICultureName** - Contains the language code for the UI culture in which the help files are written.
- **UICultureVersion** - Contains a 4-part version number in "N1.N2.N3.N4" format that represents the version of the help CAB file in the UI culture. Increment this version number whenever you upload new help CAB files in the UI culture that's specified by **UICultureName**. For more information about this value, see [Version Class](#).

Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

HelpInfo XML Sample File

Article • 07/10/2023

This topic displays a sample of a well-formed Updatable Help Information file, commonly known as "HelpInfo XML file." In this sample file, the UI culture elements are arranged in alphabetical order by UI culture name. Alphabetical ordering is a best practice, but it's not required.

HelpInfo XML Sample File

XML

```
<?xml version="1.0" encoding="utf-8"?>
<HelpInfo xmlns="http://schemas.microsoft.com/powershell/help/2010/05">
    <HelpContentURI>https://go.microsoft.com/fwlink/?LinkID=141553</HelpContentURI>
    <SupportedUICultures>
        <UICulture>
            <UICultureName>de-DE</UICultureName>
            <UICultureVersion>2.15.0.10</UICultureVersion>
        </UICulture>
        <UICulture>
            <UICultureName>en-US</UICultureName>
            <UICultureVersion>3.2.0.7</UICultureVersion>
        </UICulture>
        <UICulture>
            <UICultureName>it-IT</UICultureName>
            <UICultureVersion>1.1.0.5</UICultureVersion>
        </UICulture>
        <UICulture>
            <UICultureName>ja-JP</UICultureName>
            <UICultureVersion>3.2.0.4</UICultureVersion>
        </UICulture>
    </SupportedUICultures>
</HelpInfo>
```

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to name a HelpInfo XML file

Article • 07/10/2023

Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This topic explains the required name format for the Updatable Help Information files, commonly known as HelpInfo XML files. A HelpInfo XML file must have a name with the following format.

```
<ModuleName>_<ModuleGUID>_HelpInfo.xml
```

The elements of the name are as follows.

- `<ModuleName>` - The value of the **Name** property of the **ModuleInfo** object that the [Get-Module](#) cmdlet returns.
- `<ModuleGUID>` - The value of the **GUID** key in the module manifest.

For example, if the module name is "TestModule" and the module GUID is 9cabb9ad-f2ac-4914-a46b-bfc1bebf07f9, the name of the HelpInfo XML file for the module would be:

```
TestModule_9cabb9ad-f2ac-4914-a46b-bfc1bebf07f9_HelpInfo.xml
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to set HelpInfo XML version numbers

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

The version numbers in a HelpInfo XML file are critical to the operation of Updatable Help. The [Update-Help](#) and [Save-Help](#) cmdlets download new help files only when the version number for a UI culture in the remote HelpInfo XML file is greater than the version number for that UI culture in the local HelpInfo XML, or there is no local HelpInfo XML file.

The HelpInfo XML file uses the 4-part version number that's defined in the **System.Version** class of the Microsoft .NET Framework. The format is `N1.N2.N3.N4`. Module authors can use any version numbering scheme that's permitted by the **System.Version** class. Updatable Help requires only that the version number for a UI culture increase when a new version of the CAB file for that UI culture is uploaded to the location that's specified by the **HelpContentURI** element in the HelpInfo XML file.

The following example shows the elements of the HelpInfo XML file for the German (de-DE) UI culture when the version is 2.15.0.10.

XML

```
<UICulture>
  <UICultureName>de-DE</UICultureName>
  <UICultureVersion>2.15.0.10</UICultureVersion>
</UICulture>
```

The version number for a UI culture reflects the version of the CAB file for that UI culture. The version number applies to the entire CAB file. You can't set different version numbers for different files in the CAB file. The version number for each UI culture is evaluated independently and need not be related to the version numbers for other UI cultures that the module supports.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to prepare Updatable Help CAB files

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This topic explains the contents and use of cabinet files in Windows PowerShell Updatable Help.

This section includes the following topics.

- [How to Create and Upload CAB Files](#)
- [How to Name an Updatable Help CAB File](#)
- [File Types Permitted in an Updatable Help CAB File](#)

See also

[Supporting Updatable Help](#)

ⓘ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[ⓘ Open a documentation issue](#)

[ⓘ Provide product feedback](#)

How to create and upload CAB files

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updatable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This topic explains how to create Updatable Help CAB files and upload them to the location where the Updatable Help cmdlets can find them.

How to create and upload updatable help CAB files

You can use the Updatable Help feature to deliver new or updated help files for a module in multiple languages and cultures. An Updatable Help package for a module consists of one HelpInfo XML file and one or more cabinet (.CAB) files. Each CAB file contains help files for the module in one UI culture. Use the following procedure to create CAB files for Updatable Help.

1. Organize the help files for the module by UI culture. Each Updatable Help CAB file contains the help files for one module in one UI culture. You can deliver multiple help CAB files for the module, each for a different UI culture.
2. Verify that help files include only the file types permitted for Updatable Help and validate them against a help file schema. If the `Update-Help` cmdlet encounters a file that's invalid or is not a permitted type, it doesn't install the invalid file and stops installing files from the CAB. For a list of permitted file types, see [File Types Permitted in an Updatable Help CAB File](#).
3. Include all the help files for the module in the UI culture, not only files that are new or have changed. If the CAB file is incomplete, users who download help files for the first time or do not download every update, won't have all the help files.
4. Use a utility that creates cabinet files, such as `MakeCab.exe`. PowerShell doesn't include cmdlets that create CAB files.

5. Name the CAB files. For more information, see [How to Name an Updatable Help CAB File](#).

6. Upload the CAB files for the module to the location that's specified by the **HelpContentUri** element in the HelpInfo XML file for the module. Then upload the HelpInfo XML file to the location that's specified by the **HelpInfoUri** key of the module manifest. The **HelpContentUri** and **HelpInfoUri** can point to the same location.

 **Caution**

The value of the **HelpInfoUri** key and the **HelpContentUri** element must begin with `http` or `https`. The value must a URL path pointing to the location (folder) containing the updateable help. The URL must end with `/`. The URL must not include a filename.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to name an Updatable Help CAB file

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

An updatable cabinet (.CAB) file must have a name with the following format.

```
<ModuleName>_<ModuleGUID>_<UICulture>_HelpContent.cab
```

The elements of the name are as follows.

- `<ModuleName>` -The value of the **Name** property of the **ModuleInfo** object that the [Get-Module](#) cmdlet returns.
- `<ModuleGUID>` - The value of the **GUID** key in the module manifest.
- `<UICulture>` - The UI culture of the help files in the CAB file. This value must match the value of one of the **UICulture** elements in the HelpInfo XML file for the module.

For example, if the module name is "TestModule," the module GUID is 9cabb9ad-f2ac-4914-a46b-bfc1bebf07f9, and the UI culture is `en-US`, the name of the CAB file would be:

```
TestModule_9cabb9ad-f2ac-4914-a46b-bfc1bebf07f9_en-US_HelpContent.cab
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

File Types Permitted in an Updatable Help CAB File

Article • 07/10/2023

Uncompressed CAB file content is limited to 1 GB by default. To bypass this limit, users have to use the **Force** parameter of the [Update-Help](#) and [Save-Help](#) cmdlets.

To assure the security of help files that are downloaded from the internet, an Updatable Help CAB file can include only the file types listed below. The [Update-Help](#) cmdlet validates all files against the help topic schemas. If the [Update-Help](#) cmdlet encounters a file that's invalid or is not a permitted type, it doesn't install the invalid file and stops installing files from the CAB on the user's computer.

- XML-based help topics for cmdlets.
- XML-based help topics for scripts and functions.
- XML-based help topics for PowerShell providers.
- Text-based help topics, such as About topics.

The [Update-Help](#) verifies the CAB contents when it unpacks the CAB. If [Update-Help](#) finds non-compliant file types in an Updatable Help CAB file, it generates a terminating error and stops the operation. It doesn't install any help files from the CAB, even those of compliant file types.

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

ⓘ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

ⓘ Open a documentation issue

more information, see [our contributor guide](#).

 [Provide product feedback](#)

How to update help files

Article • 07/10/2023

Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

There are many reasons to update help files, such as correcting errors, clarifying a concept, answering a frequently asked question, adding new files, or adding new and better examples.

To update a help file:

1. Change the files.
2. Translate the files into other UI cultures.
3. Collect all help files (new, changed, and unchanged) for the module in each UI culture.
4. Validate the files against the XML schema.
5. Rebuild the CAB files for each UI culture.
6. In the HelpInfo XML file, increment the version numbers of the CAB file for each UI culture.
7. Upload the new CAB files to the location that's specified by the value of the **HelpContentUri** element in the HelpInfo XML file. Replace the older CAB files with the new CAB files.
8. Upload the updated HelpInfo XML file to the location that's specified by the **HelpInfoUri** key in the module manifest. Replace the older HelpInfo XML file with the new file.



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

How to test Updatable Help

Article • 07/10/2023

Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updatable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This topic describes approaches to testing Updatable Help for a module.

Using verbose to detect errors

After uploading the HelpInfo XML file and CAB files for your module, test the files by running an [Update-Help](#) command with the **Verbose** parameter. The **Verbose** parameter directs [Update-Help](#) to report the critical steps in its actions, from reading the **HelpInfoUri** key in the module manifest to validating the file types in the unpacked CAB file and placing the files in the language-specific module directory.

When all verbose messages are resolved, run an [Update-Help](#) command with the **Debug** parameter. This parameter should detect any remaining problems with the Updatable Help files.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Supporting Online Help

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

Beginning in PowerShell 3.0, there are two ways to support the `Get-Help` Online feature for PowerShell commands. This topic explains how to implement this feature for different command types.

About Online Help

Online help has always been a vital part of PowerShell. Although the `Get-Help` cmdlet displays help topics at the command prompt, many users prefer the experience of reading online, including color-coding, hyperlinks, and sharing ideas in Community Content and wiki-based documents. Most importantly, before the advent of Updatable Help, online help provided the most up-to-date version of the help files.

With the advent of Updatable Help in PowerShell 3.0, online help still plays a vital role. In addition to the flexible user experience, online help provides help to users who don't or can't use Updatable Help to download help topics.

How Get-Help -Online Works

To help users find the online help topics for commands, the `Get-Help` command has an `Online` parameter that opens the online version of help topic for a command in the user's default internet browser.

For example, the following command opens the online help topic for the `Invoke-Command` cmdlet.

PowerShell

```
Get-Help Invoke-Command -Online
```

To implement `Get-Help -Online`, the `Get-Help` cmdlet looks for a Uniform Resource Identifier (URI) for the online version help topic in the following locations.

- The first link in the **Related Links** section of the help topic for the command. The help topic must be installed on the user's computer. This feature was introduced in PowerShell 2.0.
- The **HelpUri** property of any command. The **HelpUri** property is accessible even when the help topic for the command isn't installed on the user's computer. This feature was introduced in PowerShell 3.0.

`Get-Help` looks for a URI in the first entry in the **Related Links** section before getting the **HelpUri** property value. If the property value is incorrect or has changed, you can override it by entering a different value in the first related link. However, the first related link works only when the help topics are installed on the user's computer.

Adding a URI to the first related link of a command help topic

You can support `Get-Help -Online` for any command by adding a valid URI to the first entry in the **Related Links** section of the XML-based help topic for the command. This option is valid only in XML-based help topics and works only when the help topic is installed on the user's computer. When the help topic is installed and the URI is populated, this value takes precedence over the **HelpUri** property of the command.

To support this feature, the URI must appear in the `maml:uri` element under the first `maml:relatedLinks/maml:navigationLink` element in the `maml:relatedLinks` element.

The following XML shows the correct placement of the URI. The `Online version:` text in the `maml:linkText` element is a best practice, but it's not required.

XML

```
<maml:relatedLinks>
  <maml:navigationLink>
    <maml:linkText>Online version:</maml:linkText>
    <maml:uri>https://go.microsoft.com/fwlink/?LinkID=113279</maml:uri>
  </maml:navigationLink>
  <maml:navigationLink>
    <maml:linkText>about_History</maml:linkText>
    <maml:uri/>
```

```
</maml:navigationLink>  
</maml:relatedLinks>
```

Adding the HelpUri property to a command

This section shows how to add the **HelpUri** property to commands of different types.

Adding a HelpUri Property to a Cmdlet

For cmdlets written in C#, add a **HelpUri** attribute to the **Cmdlet** class. The value of the attribute must be a URI that begins with `http` or `https`.

The following code shows the **HelpUri** attribute of the `Get-History` cmdlet class.

```
C#
```

```
[Cmdlet(VerbsCommon.Get, "History", HelpUri =  
"https://go.microsoft.com/fwlink/?LinkID=001122")]
```

Adding a HelpUri property to an advanced function

For advanced functions, add a **HelpUri** property to the **CmdletBinding** attribute. The value of the property must be a URI that begins with "http" or "https".

The following code shows the **HelpUri** attribute of the `New-Calendar` function

```
PowerShell
```

```
function New-Calendar {  
    [CmdletBinding(SupportsShouldProcess=$true,  
    HelpUri="https://go.microsoft.com/fwlink/?LinkID=01122")]
```

Adding a HelpUri attribute to a cim command

For CIM commands, add a **HelpUri** attribute to the **CmdletMetadata** element in the CDXML file. The value of the attribute must be a URI that begins with `http` or `https`.

The following code shows the **HelpUri** attribute of the `Start-Debug` CIM command

```
XML
```

```
<CmdletMetadata Verb="Debug" HelpUri="https://go.microsoft.com/fwlink/?LinkID=001122"/>
```

Adding a HelpUri attribute to a workflow

For workflows that are written in the PowerShell language, add an `.EXTERNALHELP` comment keyword to the workflow code. The value of the keyword must be a URI that begins with `http` or `https`.

ⓘ Note

The `HelpUri` property isn't supported for XAML-based workflows in PowerShell.

The following code shows the `.EXTERNALHELP` keyword in a workflow file.

PowerShell

```
# .EXTERNALHELP "https://go.microsoft.com/fwlink/?LinkID=138338"
```

How to add dynamic parameters to a provider help topic

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This section explains how to populate the **DYNAMIC PARAMETERS** section of a provider help topic.

Dynamic parameters are parameters of a cmdlet or function that are available only under specified conditions.

The dynamic parameters that are documented in a provider help topic are the dynamic parameters that the provider adds to the cmdlet or function when the cmdlet or function is used in the provider drive.

Dynamic parameters can also be documented in custom cmdlet help for a provider. When writing both provider help and custom cmdlet help for a provider, include the dynamic parameter documentation in both documents.

If a provider doesn't implement any dynamic parameters, the provider help topic contains an empty `DynamicParameters` element.

To add dynamic parameters

1. In the `<AssemblyName>.dll-help.xml` file, within the `providerHelp` element, add a `DynamicParameters` element. The `DynamicParameters` element should appear after the `Tasks` element and before the `RelatedLinks` element.

For example:

XML

```
<providerHelp>
  <Tasks>
```

```
</Tasks>
<DynamicParameters>
</DynamicParameters>
<RelatedLinks>
</RelatedLinks>
</providerHelp>
```

If the provider doesn't implement any dynamic parameters, the `DynamicParameters` element can be empty.

2. Within the `DynamicParameters` element, for each dynamic parameter, add a `DynamicParameter` element.

For example:

XML

```
<DynamicParameters>
  <DynamicParameter>
  </DynamicParameter>
</DynamicParameters>
```

3. In each `DynamicParameter` element, add a `Name` and `CmdletSupported` element.

- Name - Specifies the parameter name
- CmdletSupported - Specifies the cmdlets in which the parameter is valid.
Type a comma-separated list of cmdlet names.

For example, the following XML documents the `Encoding` dynamic parameter that the Windows PowerShell FileSystem provider adds to the `Add-Content`, `Get-Content`, `Set-Content` cmdlets.

XML

```
<DynamicParameters>
  <DynamicParameter>
    <Name> Encoding </Name>
    <CmdletSupported> Add-Content, Get-Content, Set-Content
  </CmdletSupported>
</DynamicParameters>
```

4. In each `DynamicParameter` element, add a `Type` element. The `Type` element is a container for the `Name` element which contains the .NET type of the value of the dynamic parameter.

For example, the following XML shows that the .NET type of the `Encoding` dynamic parameter is the [FileSystemCmdletProviderEncoding](#) enumeration.

```
XML

<DynamicParameters/>
  <DynamicParameter>
    <Name> Encoding </Name>
    <CmdletSupported> Add-Content, Get-Content, Set-Content
  </CmdletSupported>
  <Type>
    <Name>
      Microsoft.PowerShell.Commands.FileSystemCmdletProviderEncoding </Name>
    <Type>
    ...
  </DynamicParameters>
```

5. Add the `Description` element, which contains a brief description of the dynamic parameter. When composing the description, use the guidelines prescribed for all cmdlet parameters in [How to Add Parameter Information](#).

For example, the following XML includes the description of the `Encoding` dynamic parameter.

```
XML

<DynamicParameters/>
  <DynamicParameter>
    <Name> Encoding </Name>
    <CmdletSupported> Add-Content, Get-Content, Set-Content
  </CmdletSupported>
  <Type>
    <Name>
      Microsoft.PowerShell.Commands.FileSystemCmdletProviderEncoding </Name>
    <Type>
      <Description> Specifies the encoding of the output file that
      contains the content. </Description>
    ...
  </DynamicParameters>
```

6. Add the `PossibleValues` element and its child elements. Together, these elements describe the values of the dynamic parameter. This element is designed for enumerated values. If the dynamic parameter doesn't take a value, such as is the case with a switch parameter, or the values can't be enumerated, add an empty `PossibleValues` element.

The following table lists and describes the `PossibleValues` element and its child elements.

- `PossibleValues` - This element is a container. Its child elements are described below. Add one `PossibleValues` element to each provider help topic. The element can be empty.
- `PossibleValue` - This element is a container. Its child elements are described below. Add one `PossibleValue` element for each value of the dynamic parameter.
- `Value` - Specifies the value name.
- `Description` - This element contains a `Para` element. The text in the `Para` element describes the value that's named in the `Value` element.

For example, the following XML shows one `PossibleValue` element of the `Encoding` dynamic parameter.

XML

```
<DynamicParameters/>
  <DynamicParameter>
    ...
      <Description> Specifies the encoding of the output file that
contains the content. </Description>
      <PossibleValues>
        <PossibleValue>
          <Value> ASCII </Value>
          <Description>
            <para> Uses the encoding for the ASCII (7-bit)
character set. </para>
            </Description>
          </PossibleValue>
        ...
      </PossibleValues>
  </DynamicParameter>
```

Example

The following example shows the `DynamicParameters` element of the `Encoding` dynamic parameter.

XML

```
<DynamicParameters/>
  <DynamicParameter>
    <Name> Encoding </Name>
    <CmdletSupported> Add-Content, Get-Content, Set-Content
```

```
</CmdletSupported>
<Type>
  <Name>
Microsoft.PowerShell.Commands.FileSystemCmdletProviderEncoding </Name>
  <Type>
    <Description> Specifies the encoding of the output file that
contains the content. </Description>
    <PossibleValues>
      <PossibleValue>
        <Value> ASCII </Value>
        <Description>
          <para> Uses the encoding for the ASCII (7-bit) character
set. </para>
        </Description>
      </PossibleValue>
      <PossibleValue>
        <Value> Unicode </Value>
        <Description>
          <para> Encodes in UTF-16 format using the little-endian
byte order. </para>
        </Description>
      </PossibleValue>
    </PossibleValues>
  </Type>
</DynamicParameters>
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to Add a See Also Section to a Provider Help Topic

Article • 07/10/2023

ⓘ Note

Manual authoring of XML-based help is very difficult. The **PlatyPS** module allows you to write help in Markdown and then convert it to XML-based help. This makes it much easier to write and maintain help. **PlatyPS** can also create the Updateable Help packages for you. For more information, see [Create XML-based help using PlatyPS](#).

This section explains how to populate the **SEE ALSO** section of a provider help topic.

The **SEE ALSO** section consists of a list of topics that are related to the provider or might help the user better understand and use the provider. The topic list can include cmdlet help, provider help and conceptual ("about") help topics in Windows PowerShell. It can also include references to books, paper, and online topics, including an online version of the current provider help topic.

When you refer to online topics, provide the URI or a search term in plain text. The `Get-Help` cmdlet doesn't link or redirect to any of the topics in the list. Also, the `Online` parameter of the `Get-Help` cmdlet doesn't work with provider help.

The **See Also** section is created from the `RelatedLinks` element and the tags that it contains. The following XML shows how to add the tags.

To Add SEE ALSO Topics

1. In the `<AssemblyName>.dll-help.xml` file, within the `providerHelp` element, add a `RelatedLinks` element. The `RelatedLinks` element should be the last element in the `providerHelp` element. Only one `RelatedLinks` element is permitted in each provider help topic.

For example:

XML

```
<providerHelp>
  <RelatedLinks>
```

```
</RelatedLinks>  
</providerHelp>
```

2. For each topic in the SEE ALSO section, within the `RelatedLinks` element, add a `navigationLink` element. Then, within each `navigationLink` element, add one `linkText` element and one `uri` element. If you aren't using the `uri` element, you can add it as an empty element (`<uri/>`).

For example:

XML

```
<providerHelp>  
  <RelatedLinks>  
    <navigationLink>  
      <linkText> </linkText>  
      <uri> </uri>  
    </navigationLink>  
  </RelatedLinks>  
</providerHelp>
```

3. Type the topic name between the `linkText` tags. If you are providing a URI, type it between the `uri` tags. To indicate the online version of the current provider help topic, between the `linkText` tags, type "Online version:" instead of the topic name. Typically, the "Online version:" link is the first topic in the SEE ALSO topic list.

The following example include three SEE ALSO topics. The first refer to the online version of the current topic. The second refers to a Windows PowerShell cmdlet help topic. The third refers to another online topic.

XML

```
<providerHelp>  
  <RelatedLinks>  
    <navigationLink>  
      <linkText> Online version: </linkText>  
      <uri>http://www.fabrikam.com/help/myFunction.htm</uri>  
    </navigationLink>  
    <navigationLink>  
      <linkText> about_functions </linkText>  
      <uri/>  
    </navigationLink>  
    <navigationLink>  
      <linkText> Windows PowerShell Getting Started Guide  
</linkText>  
      <uri>https://go.microsoft.com/fwlink/?LinkID=89597</uri>  
    </navigationLink>
```

```
</RelatedLinks>  
</providerHelp>
```

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Extended Type System Overview

Article • 12/18/2023

PowerShell uses its **PSObject** object to extend the types of objects in two ways. First, the **PSObject** object provides a way to show different views of specific object types. This is referred to as showing an adapted view of an object. Second, the **PSObject** object provides a way to add members to existing object. Together, by wrapping an existing object, referred to as the base object, the **PSObject** object provides an extended type system (ETS) that script and cmdlet developers can use to manipulate .NET objects within the shell.

Cmdlet and Script Development Issues

ETS resolves two fundamental issues:

First, some .NET Objects do not have the necessary default behavior for acting as the data between cmdlets.

- Some .NET objects are "meta" objects (for example: WMI Objects, ADO objects, and XML objects) whose members describe the data they contain. However, in a scripting environment it is the contained data that is most interesting, not the description of the contained data. ETS resolves this issue by introducing the notion of Adapters that adapt the underlying .NET object to have the expected default semantics.
- Some .NET Object members are inconsistently named, provide an insufficient set of public members, or provide insufficient capability. ETS resolves this issue by introducing the ability to extend the .NET object with additional members.

Second, the PowerShell scripting language is *typeless* in that a variable does not need to be declared of a particular type. That is, the variables a script developer creates are by nature *typeless*. However, the PowerShell system is "type-driven" in that it depends on having a type name to operate against for basic operations such as outputting results or sorting.

Therefore a script developer must have the ability to state the type of one of their variables and build up their own set of dynamically typed "objects" that contain properties and methods and can participate in the type-driven system. ETS solves this problem by providing a common object for the scripting language that has the ability to state its type dynamically and to add members dynamically.

Fundamentally, ETS resolves the issue mentioned previously by providing the **PSObject** object, which acts as the basis of all object access from the scripting language and provides a standard abstraction for the cmdlet developer.

Cmdlet Developers

For the cmdlet developers, ETS provides the following support:

- The abstractions to work against objects in a generic way using the **PSObject** object. ETS also provides the ability to drill past these abstractions if required.
- The mechanisms to create a default behavior for formatting, sorting, serialization, and other system manipulations of their object type using a well-known set of extended members.
- The means to operate against any object using the same semantics as the script language using a **LanguagePrimitives** object.
- The means to dynamically "type" a hash table so that the rest of the system can operate against it effectively.

Script Developers

For the script developers, ETS provides the following support:

- The ability to reference any underlying object type using the same syntax (`$a.x`).
- The ability to access beyond the abstraction provided by the **PSObject** object (such as accessing only adapted members, or accessing the base object itself).
- The ability to define well-known members that control the formatting, sorting, serialization, and other manipulations of an object instance or type.
- The means to name an object as a specific type and thus control the inheritance of its type-based members.
- The ability to add, remove, and modify extended members.
- The ability to manipulate the **PSObject** object itself if required.

The **PSObject** class

The **PSObject** object is the basis of all object access from the scripting language and provides a standard abstraction for the cmdlet developer. It contains a base-object (a .NET object) and any instance members (members, specifically extended members, that are present on a particular object instance while not necessarily on other objects of the same type). Depending on the type of the base-object, the **PSObject** object might also provide implicit and explicit access to adapted members as well as any type-based extended members.

The **PSObject** object provides the following mechanisms:

- The ability to construct a **PSObject** with or without a base-object.
- The ability to access of all members of each constructed **PSObject** object through a common lookup algorithm and the ability to override that algorithm when required.
- The ability to get and set the type-names of the constructed **PSObject** objects so that scripts and cmdlets can reference similar **PSObject** objects by the same type-name, regardless of the type of their base-object.

How to Construct a PSObject

The following list describes ways to create a **PSObject** object:

- Calling the **PSObject** `#ctor` constructor creates a new **PSObject** object with a base-object of **PSCustomObject**. A base-object of this type indicates that the **PSObject** object has no meaningful base-object. However, a **PSObject** object with this type of base-object does provide a property bag that cmdlet developers can find helpful by adding extended-members.

Developers can also specify the object type-name, which allows this object to share its extended-members with other **PSObject** objects of the same type-name.

- Calling the **PSObject** `#ctor(System.Object)` constructor creates a new **PSObject** object with a base-object of type **System.Object**.

In this case, the type-name for the created object is a collection of the derivation hierarchy of the base-object. For example, the type-name for the **PSObject** that contains a **ProcessInfo** base-object would include the following names:

- **System.Diagnostics.Process**
- **System.ComponentModel.Component**
- **System.MarshalByRefObject**
- **System.Object**

- Calling the **PSObject** `.AsPSObject(System.Object)` method creates a new **PSObject** object based on a supplied object.

If the supplied object is of type **System.Object**, the supplied object is used as the base-object for the new **PSObject** object. If the supplied object is already a **PSObject** object, the supplied object is returned as is.

Base, adapted, and extended members

Conceptually, ETS uses the following terms to show the relationship between the original members of the base-object and those members added by PowerShell. For more information about the specific types of members that are used by the **PSObject** object, see [PSObject class](#).

Base-object members

If the base-object is specified when constructing the **PSObject** objects, then the members of the base-object are made available through the Members property.

Adapted members

When a base-object is a meta-object, one that contains data in a generic fashion whose properties "describe" their contained data, ETS adapts those objects to a view that allows for direct access to the data through adapted members of the **PSObject** object. Adapted members and base-object members are accessed through the Members property.

Extended members

In addition to the members made available from the base-object or those adapted members created by PowerShell, a **PSObject** may also define extended members that extend the original base-object with additional information that is useful in the scripting environment.

For example, all the core cmdlets provided by PowerShell, such as the Get-Content and Set-Content cmdlets, take a Path parameter. To ensure that these cmdlets, and others, can work against objects of different types, a Path member can be added to those objects so that they all state their information in a common way. This extended Path member ensures that the cmdlets can work against all those types even though their base class might not have a Path member.

Extended members, adapted members, and base-object members are all accessed through the Members property.

Extended Type System class members

Article • 09/17/2021

ETS refers to a number of different kinds of members whose types are defined by the **PSMemberTypes** enumeration. These member types include properties, methods, members, and member sets that are each defined by their own CLR type. For example, a **NoteProperty** is defined by its own **PSNoteProperty** type. These individual CLR types have both their own unique properties and common properties that are inherited from the [PSMemberInfo class](#).

The PSMemberInfo class

The **PSMemberInfo** class serves as a base class for all ETS member types. This class provides the following base properties to all member CLR types.

- **Name** property: The name of the member. This name can be defined by the base-object or defined by PowerShell when adapted members or extended members are exposed.
- **Value** property: The value returned from the particular member. Each member type defines how it handles its member value.
- **TypeNameOfValue** property: This is the name of the CLR type of the value that is returned by the Value property.

Accessing members

Collections of members can be accessed through the **Members**, **Methods**, and **Properties** properties of the **PSObject** object.

ETS properties

ETS properties are members that can be treated as a property. Essentially, they can appear on the left-hand side of an expression. They include alias properties, code properties, PowerShell properties, note properties, and script properties. For more information about these types of properties, see [ETS properties](#).

ETS methods

ETS methods are members that can take arguments, may return results, and cannot appear on the left-hand side of an expression. They include code methods, PowerShell

methods, and script methods. For more information about these types of methods, see [ETS methods](#).

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ETS member sets

Article • 12/18/2023

Member sets allow you to partition the members of the **PSObject** object into two subsets so that the members of the subsets can be referenced together by their subset name. The two types of subsets include property sets and member sets. For example of how PowerShell uses member sets, there is a specific property set named **DefaultDisplayPropertySet** that is used to determine, at runtime, which properties to display for a given **PSObject** object.

Property Sets

Property sets can include any number of properties of a **PSObject** type. In general, a property set can be used whenever a collection of properties (of the same type) is needed. The property set is created by calling the

`PSPropertySet(System.String, System.Collections.Generic.IEnumerable{System.String})`

constructor with the name of the property set and the names of the referenced properties. The created **PSPropertySet** object can then be used as an alias that points to the properties in the set. The **PSPropertySet** class has the following properties and methods.

- **IsInstance** property: Gets a **Boolean** value that indicates the source of the property.
- **MemberType** property: Gets the type of properties in the property set.
- **Name** property: Gets the name of the property set.
- **ReferencedPropertyNames** property: Gets the names of the properties in the property set.
- **TypeNameOfValue** property: Gets a **PropertySet** enumeration constant that defines this set as a property set.
- **Value** property: Gets or sets the **PSPropertySet** object.
- `PSPropertySet.Copy` method: Makes an exact copy of the **PSPropertySet** object.
- `PSMemberSet.ToString` method: Converts the **PSPropertySet** object to a string.

Member Sets

Member sets can include any number of extended members of any type. The member set is created by calling the

`PSMemberSet(System.String, System.Collections.Generic.IEnumerable{System.Management.Automation.PSMemberInfo})`

constructor with the name of the member set and the names

of the referenced members. The created **PSPropertySet** object can then be used as an alias that points to the members in the set. The **PSMemberSet** class has the following properties and methods.

- **IsInstance** property: Gets a **Boolean** value that indicates the source of the member.
- **Members** property: Gets all the members of the member set.
- **MemberType** property: Gets a **MemberSet** enumeration constant that defines this set as a member set.
- **Methods** property: Gets the methods included in the member set.
- **Properties** property: Gets the properties included in the member set.
- **TypeNameOfValue** property: Gets a **MemberSet** enumeration constant that defines this set as a member set.
- **Value** property: Gets the **PSMemberSet** object.
- **PSMemberSet.Copy** method: Makes an exact copy of the **PSMemberSet** object.
- **PSMemberSet.ToString** method: Converts the **PSMemberSet** object to a string.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ETS properties

Article • 09/17/2021

Properties are members that can be treated as a property. Essentially, they can appear on the left-hand side of an expression. The properties that are available include alias, code, note, and script properties.

Alias Property

An alias property is a property that references another property that the **PSObject** object contains. It is used primarily to rename the referenced property. However, it may also be used to convert the referenced property's value to another type. With respect to ETS, this type of property is always an extended-member and is defined by the [PSAliasProperty](#) class. The class includes the following properties.

- **ConversionType** property: The CLR type used to convert the referenced member's value.
- **IsGettable** property: Indicates whether the value of the referenced property can be retrieved. This property is dynamically determined by examining the **IsGettable** property of the referenced property.
- **IsSettable** property: Indicates whether the value of the referenced property can be set. This property is dynamically determined by examining the **IsSettable** property of the referenced property.
- **MemberType** property: An **AliasProperty** enumeration constant that defines this property as an alias property.
- **ReferencedMemberName** property: The name of the referenced property that this alias refers to.
- **TypeNameOfValue** property: The full name of the CLR type of the referenced property's value.
- **Value** property: The value of the referenced property.

Code Property

A code property is a property that is a getter and setter that is defined in a CLR language. In order for a code property to become available, a developer must write the property in some CLR language, compile, and ship the resultant assembly. This assembly must be available in the runspace where the code property is desired. With respect to ETS, this type of property is always an extended-member and is defined by the [PSCodeProperty](#) class. The class includes the following properties.

- **GetterCodeReference** property: The method used to get the value of the code property.
- **IsGettable** property: Indicates whether the value of the code property can be retrieved, that the **SetterCodeReference** property: The method used to set the value of the code property.
- **IsSettable** property: Indicates whether the value of the code property can be set, that the **SetterCodeReference** property is not null.
- **MemberType** property: A **CodeProperty** enumeration constant that defines this property as a code property.
- **SetterCodeReference** property: The method used to get the value of the code property.
- **TypeNameOfValue** property: The CLR type of the code property value that is returned by the properties get operation.
- **Value** property: The value of the code property. When this property is retrieved, the getter code in the **GetterCodeReference** property is invoked, passing the current **PSObject** object and returning the value returned by the invocation. When this property is set, the setter code in the **SetterCodeReference** property is invoked, passing the current **PSObject** object as the first argument and the object used to set the value as the second argument.

Note Property

A Note property is a property that has a name/value pairing. With respect to ETS, this type of property is always an extended-member and is defined by the [PSNoteProperty](#) class. The class includes the following properties.

- **IsGettable** property: Indicates whether the value of the note property can be retrieved.
- **IsSettable** property: Indicates whether the value of the note property can be set.
- **MemberType** property: A **NoteProperty** enumeration constant that defines this property as a note property.
- **TypeNameOfValue** property: The fully-qualified type name of the object returned by the note property's get operation.
- **Value**: The value of the note property.

PowerShell property

A PowerShell property is a property defined on the base object or a property that is made available through an adapter. It can refer to both CLR fields as well as CLR properties. With respect to ETS, this type of property can be either a base-member or an

adapter-member and is defined by the [PSProperty](#) class. The class includes the following properties.

- **IsGettable** property: Indicates whether the value of the base or adapted property can be retrieved.
- **IsSettable** property: Indicates whether the value of the base or adapted property can be set.
- **MemberType** property: A Property enumeration constant that defines this property as a PowerShell property.
- **TypeNameOfValue** property: The fully-qualified name of the property value type. For example, for a property whose value is a string, its property value type is `System.String`.
- **Value** property: The value of the property. If the get or set operation is called on a property that does not support that operation, a `GetValueException` or `SetValueException` exception is thrown

PowerShell Script property

A Script property is a property that has getter and setter scripts. With respect to ETS, this type of property is always an extended-member and is defined by the [PSScriptProperty](#) class. The class includes the following properties.

- **GetterScript** property: The script used to retrieve the script property value.
- **IsGettable** property: Indicates whether the **GetterScript** property exposes a script block.
- **IsSettable** property: Indicates whether the **SetterScript** property exposes a script block.
- **MemberType** property: A ScriptProperty enumeration constant that identifies this property as a script property.
- **SetterScript** property: The script used to set the script property value.
- **TypeNameOfValue** property: The fully-qualified type name of the object returned by the getter script. In this case `System.Object` is always returned.
- **Value** property: The value of the script property. A get invokes the getter script and returns the value provided. A set invokes the setter script.

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

ETS class methods

Article • 09/17/2021

ETS methods are members that can take arguments, may return results, and cannot appear on the left-hand side of an expression. The methods that are available within ETS include code, Windows PowerShell, and script methods.

ⓘ Note

From scripts, methods are accessed using the same syntax as other members with the addition of parenthesis at the end of the method name.

Code Methods

A code method is an extended member that is defined in a CLR language. It provides similar functionality to a method defined on a base object; however, a code method may be added dynamically to an **PSObject** object. In order for a code method to become available, a developer must write the property in some CLR language, compile, and ship the resultant assembly. This assembly must be available in the runspace where the code method is desired. Be aware that a code method implementation must be thread safe. Access to these methods is done through **PSCodemethod** objects that provides the following public methods and properties.

- **PSCodemethod.Copy** method: Makes an exact copy of the **PSCodemethod** object.
- **PSCodemethod.Invoke(System.Object[])** method: Invokes the underlying code method.
- **PSCodemethod.ToString** method: Converts the **PSCodemethod** object to a string.
- **PSCodemethod.CodeReference** property: Gets the underlying method that the code method is based on.
- **PSMemberInfo.Isinstance** property: Gets a **Boolean** value that indicates the source of the member.
- **PSCodemethod.MemberType** property: Gets an **PSMemberTypes.CodeMethod** enumeration constant that identifies this method as a code method.
- **PSMemberInfo.Name** property: Gets the name of the underlying code method.
- **PSCodemethod.OverloadDefinitions** property: Gets a definition of all the overloads of the underlying code method.
- **PSCodemethod.TypeNameOfValue** property: Gets the full name of the code method.
- **PSMemberInfo.Value** property: Gets the **PSCodemethod** object.

Windows PowerShell Methods

A PowerShell method is a CLR method defined on the base object or is made accessible through an adapter. Access to these methods is done through **PSMethod** objects that provides the following public methods and properties.

- **PSMethod.Copy** method: Makes an exact copy of the **PSMethod** object.
- **PSMethod.Invoke(System.Object[])** method: Invokes the underlying method.
- **PSMethod.ToString** method: Converts the **PSMethod** object to a string.
- **PSMethodInfo.IsInstance** property: Gets a **Boolean** value that indicates the source of the member.
- **PSMethod.MemberType** property: Gets an **PSMemberTypes.Method** enumeration constant that identifies this method as a PowerShell method.
- **PSMethodInfo.Name** property: Gets the name of the underlying method.
- **PSMethod.OverloadDefinitions** property: Gets the definitions of all the overloads of the underlying method.
- **PSMethod.TypeNameOfValue** property: Gets the ETS type of this method.
- **PSMethodInfo.Value** property: Gets the **PSMethod** object.

Script Methods

A script method is an extended member that is defined in the PowerShell language. It provides similar functionality to a method defined on a base object; however, a script method may be added dynamically to an **PSObject** object. Access to these methods is done through **PSScriptMethod** objects that provides the following public methods and properties.

- **PSScriptMethod.Copy** method: Makes an exact copy of the **PSScriptMethod** object.
- **PSScriptMethod.Invoke(System.Object[])** method: Invokes the underlying script method.
- **PSScriptMethod.ToString** method: Converts the **PSScriptMethod** object to a string.
- **PSMethodInfo.IsInstance** property: Gets a **Boolean** value that indicates the source of the member.
- **PSScriptMethod.MemberType** property: Gets a **PSMemberTypes.ScriptMethod** enumeration constant that identifies this method as a script method.
- **PSMethodInfo.Name** property: Gets the name of the underlying code method.
- **PSScriptMethod.OverloadDefinitions** property: Gets the definitions of all the overloads of the underlying script method.
- **PSScriptMethod.TypeNameOfValue** property: Gets the ETS type of this method.
- **PSScriptMethod.Script** property: Gets the script used to invoke the method.

- **PSMemberInfo.Value** property: Gets the **PSScriptMethod** object.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

ETS type converters

Article • 09/17/2021

ETS uses two basic types of type converters when a call is made to the `LanguagePrimitives.ConvertTo(System.Object, System.Type)` method. When this method is called, PowerShell attempts to perform the type conversion using its standard PowerShell language converters or a custom converter. If PowerShell cannot perform the conversion, it throws an `PSInvalidCastException` exception.

Standard Windows PowerShell Language Converters

These standard conversions are checked before any custom conversions and cannot be overridden. The following table lists the type conversions performed by PowerShell when the `ConvertTo(System.Object, System.Type)` method is called. Be aware that references to the `valueToConvert` and `resultType` parameters refer to parameters of the `ConvertTo(System.Object, System.Type)` method.

 Expand table

From (<code>valueToConvert</code>)	To (<code>resultType</code>)	Returns
Null	String	""
Null	Char	'\0'
Null	Numeric	0 of the type specified in the <code>resultType</code> parameter.
Null	Boolean	Results of call to the <code>IsTrue(System.Object)(Null)</code> method.
Null	PSObject	New object of type <code>PSObject</code> .
Null	Non-value-type	Null.
Null	Nullable<T>	Null.
Derived Class	Base class	<code>valueToConvert</code>
Anything	Void	<code>AutomationNull.Value</code>
Anything	String	Calls <code>ToString</code> mechanism.
Anything	Boolean	<code>IsTrue(System.Object) (valueToConvert)</code>

From Anything (valueToConvert)	To PSObject (resultType)	Returns Results of call to the <code>AsPSObject(System.Object)(valueToConvert)</code> method.
Anything	Xml Document	Converts valueToConvert to string, then calls XMLDocument constructor.
Array	Array	Attempts to convert each element of the array.
Singleton	Array	<code>Array[0]</code> equals valueToConvert that is converted to the element type of the array.
IDictionary	Hash table	Results of call to <code>Hashtable(valueToConvert)</code> .
String	Char[]	<code>valueToConvert.ToCharArray</code>
String	RegEx	Results of call to <code>Regx(valueToConvert)</code> .
String	Type	Returns the appropriate type using the valueToConvert parameter to search RunspaceConfiguration.Assemblies .
String	Numeric	If valueToConvert is "", returns <code>0</code> of the resultType . Otherwise the culture "culture invariant" is used to produce a numeric value.
Integer	System.Enum	Converts the integer to the constant if the integer is defined by the enumeration. If the integer is not defined an PSInvalidCastException exception is thrown.

Custom conversions

If PowerShell cannot convert the type using a standard PowerShell language converter, it then checks for custom converters. PowerShell looks for several types of custom converters in the order described in this section. Be aware that references to the **valueToConvert** and **resultType** parameters refer to parameters of the `ConvertTo(System.Object, System.Type)` method. If a custom converter throws an exception, then no further attempt is made to convert the object and that exception is wrapped in a **PSInvalidCastException** exception which is then thrown.

PowerShell type converter

PowerShell type converters are used to convert a single type or a family of types, such as all types that derive from the **System.Enum** class. To create a PowerShell type converter you must implement an **PSTypeConverter** class and associate that implementation with the target class. There are two ways of associating the PowerShell type converter with its target class.

- Through the type configuration file
- By applying the **TypeConverterAttribute** attribute to the target class

PowerShell type converters, derived from the [PSTypeConverter](#) abstract class, provide methods for converting an object to a specific type or from a specific type. If the **valueToConvert** parameter contains an object that has a PowerShell Type converter associated with it, PowerShell calls the `PSTypeConverter.ConvertTo(System.Object, System.Type, System.IFormatProvider, System.Boolean)` method of the associated converter to convert the object to the type specified by the **resultType** parameter. If the **resultType** parameter references a type that has a PowerShell type converter associated with it, PowerShell calls the `PSTypeConverter.ConvertFrom(System.Object, System.Type, System.IFormatProvider, System.Boolean)` method of the associated converter to convert the object from the type specified by the **resultType** parameter.

System type converter

System type converters are used to convert a specific target class. This type of converter cannot be used to convert a family of classes. To create an system type converter you must implement an [TypeConverter](#) class and associate that implementation with the target class. There are two ways of associating the system type converter with its target class.

- Through the type configuration file
- By applying the **TypeConverterAttribute** attribute to the target class

Parse converter

If the **valueToConvert** parameter is a string, and the object type of the **resultType** parameter has a `Parse` method, then the `Parse` method is called to convert the string.

Constructor converter

If the object type of the **resultType** parameter has a constructor that has a single parameter that is the same type as the object of the **valueToConvert** parameter, then this constructor is called.

Implicit cast operator converter

If the `valueToConvert` parameter has an implicit cast operator that converts to `resultType`, then its cast operator is called. If the `resultType` parameter has an implicit cast operator that converts from `valueToConvert`, then its cast operator is called.

Explicit cast operator converter

If the `valueToConvert` parameter has an explicit cast operator that converts to `resultType`, then its cast operator is called. If the `resultType` parameter has an explicit cast operator that converts from `valueToConvert`, then its cast operator is called.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Errors and exceptions in the Extended Type System

Article • 09/17/2021

Errors can occur in ETS during the initialization of type data and when accessing a member of an **PSObject** object or using one of the utility classes such as **LanguagePrimitives**.

Runtime errors

With one exception, when casting, all exceptions thrown from ETS during runtime are either an **ExtendedTypeSystemException** exception or an exception derived from the **ExtendedTypeSystemException** class. This allows script developers to trap these exceptions using the `trap` statement in their script.

Errors getting member values

All errors that occur when getting the value of an ETS member (property, method, or parameterized property) cause a **GetValueException** or **GetValueInvocationException** exception to be thrown. When ETS recognizes that an error occurred a **GetValueException** exception is thrown. When the underlying getter of a referenced member recognizes that an error occurred, a **GetValueInvocationException** exception is thrown that may or may not include the inner exception that caused the get invocation error.

Errors setting member values

All errors that occur when setting the value of an ETS property cause a **SetValueException** or **SetValueInvocationException** exception to be thrown. When ETS recognizes that an error occurred a **SetValueException** exception is thrown. When the underlying setter of a referenced property recognizes that an error occurred, a **SetValueInvocationException** exception is thrown that may or may not include the inner exception that caused the set invocation error.

Errors invoking a method

All errors that occur when invoking an ETS method cause a **MethodException** or **MethodInvocationException** exception to be thrown. When ETS recognizes that an error occurred a **MethodException** exception is thrown. When the referenced method recognizes that an error occurred, a **MethodInvocationException** exception is thrown that may or may not include the inner exception that caused the invocation error.

Casting errors

When an invalid cast is attempted, an **PSInvalidCastException** is thrown. Because this exception derives from **System.InvalidCastException**, it is not able to be directly trapped from script. Be aware that the entity attempting the cast would need to wrap **PSInvalidCastException** in an **PSRuntimeException** for this to be trappable by scripts. If an attempt is made to set the value of an **PSPropertySet**, **PSMemberSet**, **PSMethodInfo**, or a member of the **ReadOnlyPSMemberInfoCollection`1**, a **NotSupportedException** is thrown.

Common runtime errors

Any other common runtime errors that occur are of type **ExtendedTypeSystemException** exception with no additional specific exception types.

Initialization errors

Errors may occur when initializing `types.ps1xml`. Typically, these errors are displayed when the PowerShell runtime starts. However, they can also be displayed when a module is loaded.

Windows PowerShell Programmer's Guide

Article • 09/17/2021

This programmer's guide is targeted at developers who are interested in providing a command-line management environment for system administrators. Windows PowerShell provides a simple way for you to build management commands that expose .NET objects, while allowing Windows PowerShell to do most of the work for you.

In traditional command development, you are required to write a parameter parser, a parameter binder, filters, and all other functionality exposed by each command. Windows PowerShell provides the following to make it easy for you to write commands:

- A powerful Windows PowerShell runtime (execution engine) with its own parser and a mechanism for automatically binding command parameters.
- Utilities for formatting and displaying command results using a command line interpreter (CLI).
- Support for high levels of functionality (through Windows PowerShell providers) that make it easy to access stored data.

At little cost, you can represent a .NET object by a rich command or set of commands that will offer a complete command-line experience to the administrator.

The next section covers the key Windows PowerShell concepts and terms. Familiarize yourself with these concepts and terms before starting development.

About Windows PowerShell

Windows PowerShell defines several types of commands that you can use in development. These commands include: functions, filters, scripts, aliases, and executables (applications). The main command type discussed in this guide is a simple, small command called a "cmdlet". Windows PowerShell furnishes a set of cmdlets and fully supports cmdlet customization to suit your environment. The Windows PowerShell runtime processes all command types just as it does cmdlets, using pipelines.

In addition to commands, Windows PowerShell supports various customizable Windows PowerShell providers that make available specific sets of cmdlets. The shell operates within the Windows PowerShell-provided host application (`powershell.exe`), but it is equally accessible from a custom host application that you can develop to meet specific requirements. For more information, see [How Windows PowerShell Works](#).

Windows PowerShell Cmdlets

A cmdlet is a lightweight command that is used in the Windows PowerShell environment. The Windows PowerShell runtime invokes these cmdlets within the context of automation scripts that are provided at the command line, and the Windows PowerShell runtime also invokes them programmatically through Windows PowerShell APIs.

For more information about cmdlets, see [Writing a Windows PowerShell Cmdlet](#).

Windows PowerShell Providers

In performing administrative tasks, the user may need to examine data stored in a data store (for example, the file system, the Windows Registry, or a certificate store). To make these operations easier, Windows PowerShell defines a module called a Windows PowerShell provider that can be used to access a specific data store, such as the Windows Registry. Each provider supports a set of related cmdlets to give the user a symmetrical view of the data in the store.

Windows PowerShell provides several default Windows PowerShell providers. For example, the Registry provider supports navigation and manipulation of the Windows Registry. Registry keys are represented as items, and registry values are treated as properties.

If you expose a data store that the user will need to access, you might need to write your own Windows PowerShell provider, as described in [Creating Windows PowerShell Providers](#). For more information about Windows PowerShell providers, see [How Windows PowerShell Works](#).

Host Application

Windows PowerShell includes the default host application powershell.exe, which is a console application that interacts with the user and hosts the Windows PowerShell runtime using a console window.

Only rarely will you need to write your own host application for Windows PowerShell, although customization is supported. One case in which you might need your own application is when you have a requirement for a GUI interface that is richer than the interface provided by the default host application. You might also want a custom application when you are basing your GUI on the command line. For more information, see [How to Create a Windows PowerShell Host Application](#).

Windows PowerShell Runtime

The Windows PowerShell runtime is the execution engine that implements command processing. It includes the classes that provide the interface between the host application and Windows PowerShell commands and providers. The Windows PowerShell runtime is implemented as a runspace object for the current Windows PowerShell session, which is the operational environment in which the shell and the commands execute. For operational details, see [How Windows PowerShell Works](#).

Windows PowerShell Language

The Windows PowerShell language provides scripting functions and mechanisms to invoke commands. For complete scripting information, see the Windows PowerShell Language Reference shipped with Windows PowerShell.

Extended Type System (ETS)

Windows PowerShell provides access to a variety of different objects, such as .NET and XML objects. As a consequence, to present a common abstraction for all object types the shell uses its extended type system (ETS). Most ETS functionality is transparent to the user, but the script or .NET developer uses it for the following purposes:

- Viewing a subset of the members of specific objects. Windows PowerShell provides an "adapted" view of several specific object types.
- Adding members to existing objects.
- Access to serialized objects.
- Writing customized objects.

Using ETS, you can create flexible new "types" that are compatible with the Windows PowerShell language. If you are a .NET developer, you are able to work with objects using the same semantics as the Windows PowerShell language applies to scripting, for example, to determine if an object evaluates to `true`.

For more information about ETS and how Windows PowerShell uses objects, see [Windows PowerShell Object Concepts](#).

Programming for Windows PowerShell

Windows PowerShell defines its code for commands, providers, and other program modules using the .NET Framework. You are not confined to the use of Microsoft Visual Studio in creating customized modules for Windows PowerShell, although the samples provided in this

guide are known to run in this tool. You can use any .NET language that supports class inheritance and the use of attributes. In some cases, Windows PowerShell APIs require the programming language to be able to access generic types.

Programmer's Reference

For reference when developing for Windows PowerShell, see the [Windows PowerShell SDK](#).

Getting Started Using Windows PowerShell

For more information about starting to use the Windows PowerShell shell, see the [Getting Started with Windows PowerShell](#) shipped with Windows PowerShell. A Quick Reference tri-fold document is also supplied as a primer for cmdlet use.

Contents of This Guide

 Expand table

Topic	Definition
How to Create a Windows PowerShell Provider	This section describes how to build a Windows PowerShell provider for Windows PowerShell.
How to Create a Windows PowerShell Host Application	This section describes how to write a host application that manipulates a runspace and how to write a host application that implements its own custom host.
How to Create a Windows PowerShell Snap-in	This section describes how to create a snap-in that is used to register all cmdlets and providers in an assembly and how to create a custom snap-in.
How to Create a Console Shell	This section describes how to create a console shell that is not extensible.
Windows PowerShell Concepts	This section contains conceptual information that will help you understand Windows PowerShell from the viewpoint of a developer.

See Also

[Windows PowerShell SDK](#)

How to Create a Windows PowerShell Provider

Article • 09/17/2021

This section describes how to build a Windows PowerShell provider. A Windows PowerShell provider can be considered in two ways. To the user, the provider represents a set of stored data. For example, the stored data can be the Internet Information Services (IIS) Metabase, the Microsoft Windows Registry, the Windows file system, Active Directory, and the variable and alias data stored by Windows PowerShell.

To the developer, the Windows PowerShell provider is the interface between the user and the data that the user needs to access. From this perspective, each type of provider described in this section supports a set of specific base classes and interfaces that allow the Windows PowerShell runtime to expose certain cmdlets to the user in a common way.

Providers Provided by Windows PowerShell

Windows PowerShell provides several providers (such as the FileSystem provider, Registry provider, and Alias provider) that are used to access known data stores. For more information about the providers supplied by Windows PowerShell, use the following command to access online Help:

```
PS>Get-Help about_Providers
```

Accessing the Stored Data Using Windows PowerShell Paths

Windows PowerShell providers are accessible to the Windows PowerShell runtime and to commands programmatically through the use of Windows PowerShell paths. Most of the time, these paths are used to directly access the data through the provider. However, some paths can be resolved to provider-internal paths that allow a cmdlet to use non-Windows PowerShell application programming interfaces (APIs) to access the data. For more information about how Windows PowerShell providers operate within Windows PowerShell, see [How Windows PowerShell Works](#).

Exposing Provider Cmdlets Using Windows PowerShell Drives

A Windows PowerShell provider exposes its supported cmdlets using virtual Windows PowerShell drives. Windows PowerShell applies the following rules for a Windows PowerShell drive:

- The name of a drive can be any alphanumeric sequence.
- A drive can be specified at any valid point on a path, called a "root".
- A drive can be implemented for any stored data, not just the file system.
- Each drive keeps its own current working location, allowing the user to retain context when shifting between drives.

In This Section

The following table lists topics that include code examples that build on each other. Starting with the second topic, the basic Windows PowerShell provider can be initialized and uninitialized by the Windows PowerShell runtime, the next topic adds functionality for accessing the data, the next topic adds functionality for manipulating the data (the items in the stored data), and so on.

[] Expand table

Topic	Definition
Designing Your Windows PowerShell Provider	This topic discusses things you should consider before implementing a Windows PowerShell provider. It summarizes the Windows PowerShell provider base classes and interfaces that are used.
Creating a Basic Windows PowerShell Provider	This topic shows how to create a Windows PowerShell provider that allows the Windows PowerShell runtime to initialize and uninitialized the provider.
Creating a Windows PowerShell Drive Provider	This topic shows how to create a Windows PowerShell provider that allows the user to access a data store through a Windows PowerShell drive.
Creating a Windows PowerShell Item Provider	This topic shows how to create a Windows PowerShell provider that allows the user to manipulate the items in a data store.
Creating a Windows PowerShell Container Provider	This topic shows how to create a Windows PowerShell provider that allows the user to work on multilayer data stores.

Topic	Definition
Creating a Windows PowerShell Navigation Provider	This topic shows how to create a Windows PowerShell provider that allows the user to navigate the items of a data store in a hierarchical manner.
Creating a Windows PowerShell Content Provider	This topic shows how to create a Windows PowerShell provider that allows the user to manipulate the content of items in a data store.
Creating a Windows PowerShell Property Provider	This topic shows how to create a Windows PowerShell provider that allows the user to manipulate the properties of items in a data store.

See Also

[How Windows PowerShell Works](#)

[Windows PowerShell SDK](#)

[Windows PowerShell Programmer's Guide](#)

Designing Your Windows PowerShell Provider

Article • 03/24/2025

You should implement a Windows PowerShell provider if your product or configuration exposes a set of stored data, such as a database that the user will want to navigate or browse. Additionally, if your product provides a container, even if it is not a multilevel container, it makes sense to implement a Windows PowerShell provider. For example, you might want to implement a Windows PowerShell container provider if the cmdlet verb Copy, Move, Rename, New, or Remove makes sense as an operation on your product or configuration data.

Windows PowerShell Paths Identify Your Provider

The Windows PowerShell runtime uses Windows PowerShell paths to access the appropriate Windows PowerShell provider. When a cmdlet specifies one of these paths, the runtime knows which provider to use to access the associated data store. These paths include drive-qualified paths, provider-qualified paths, provider-direct paths, and provider-internal paths. Each Windows PowerShell provider must support one or more of these paths.

For more information about Windows PowerShell paths, see [How Windows PowerShell Works](#).

Defining a Drive-Qualified Path

To allow the user to access data located at a physical drive, your Windows PowerShell provider must support a drive-qualified path. This path starts with the drive name followed by a colon (:), for example, mydrive:\abc\bar.

Defining a Provider-Qualified Path

To allow the Windows PowerShell runtime to initialize and uninitialized the provider, your Windows PowerShell provider must support a provider-qualified path. For example, FileSystem::\\uncshare\abc\bar is the provider-qualified path for the FileSystem provider furnished by Windows PowerShell.

Defining a Provider-Direct Path

To allow remote access to your Windows PowerShell provider, it should support a provider-direct path to pass directly to the Windows PowerShell provider for the current location. For example, the registry Windows PowerShell provider can use \\server\regkeypath as a provider-direct path.

Defining a Provider-Internal Path

To allow the provider cmdlet to access data using non-Windows PowerShell application programming interfaces (APIs), your Windows PowerShell provider should support a provider-internal path. This path is indicated after the ":" in the provider-qualified path. For example, the provider-internal path for the FileSystem Windows PowerShell provider is \\uncshare\abc\bar.

Changing Stored Data

When overriding methods that modify the underlying data store, always call the [System.Management.Automation.Provider.CmdletProvider.WriteItemObject*](#) method with the most up-to-date version of the item changed by that method. The provider infrastructure determines if the item object needs to be passed to the pipeline, such as when the user specifies the -PassThru parameter. If retrieving the most up-to-date item is a costly operation (performance-wise,) you can test the Context.PassThru property to determine if you actually need to write the resulting item.

Choose a Base Class for Your Provider

Windows PowerShell provides a number of base classes that you can use to implement your own Windows PowerShell provider. When designing a provider, choose the base class, described in this section, that is most suited to your requirements.

Each Windows PowerShell provider base class makes available a set of cmdlets. This section describes the cmdlets, but it does not describe their parameters.

Using the session state, the Windows PowerShell runtime makes several location cmdlets available to certain Windows PowerShell providers, such as the `Get-Location`, `Set-Location`, `Pop-Location`, and `Push-Location` cmdlets. You can use the `Get-Help` cmdlet to obtain information about these location cmdlets.

CmdletProvider Base Class

The [System.Management.Automation.Provider.CmdletProvider](#) class defines a basic Windows PowerShell provider. This class supports the provider declaration and supplies a number of properties and methods that are available to all Windows PowerShell providers. The class is invoked by the `Get-PSProvider` cmdlet to list all available providers for a session. The implementation of this cmdlet is furnished by the session state.

 **Note**

Windows PowerShell providers are available to all Windows PowerShell language scopes.

DriveCmdletProvider Base Class

The [System.Management.Automation.Provider.DriveCmdletProvider](#) class defines a Windows PowerShell drive provider that supports operations for adding new drives, removing existing drives, and initializing default drives. For example, the [FileSystem](#) provider provided by Windows PowerShell initializes drives for all volumes that are mounted, such as hard drives and CD/DVD device drives.

This class derives from the [System.Management.Automation.Provider.CmdletProvider](#) base class. The following table lists the cmdlets exposed by this class. In addition to those listed, the `Get-PSDrive` cmdlet (exposed by session state) is a related cmdlet that is used to retrieve available drives.

 [Expand table](#)

Cmdlet	Definition
<code>New-PSDrive</code>	Creates a new drive for the session, and streams drive information.
<code>Remove-PSDrive</code>	Removes a drive from the session.

ItemCmdletProvider Base Class

The [System.Management.Automation.Provider.ItemCmdletProvider](#) class defines a Windows PowerShell item provider that performs operations on the individual items of the data store, and it does not assume any container or navigation capabilities. This class derives from the [System.Management.Automation.Provider.DriveCmdletProvider](#) base class. The following table lists the cmdlets exposed by this class.

[+] Expand table

Cmdlet	Definition
Clear-Item	Clears the current content of items at the specified location, and replaces it with the "clear" value specified by the provider. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.
Get-Item	Retrieves items from the specified location, and streams the resultant objects.
Invoke-Item	Invokes the default action for the item at the specified path.
Set-Item	Sets an item at the specified location with the indicated value. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.
Resolve-Path	Resolves the wildcards for a Windows PowerShell path, and streams path information.
Test-Path	Tests for the specified path, and returns <code>true</code> if it exists and <code>false</code> otherwise. This cmdlet is implemented to support the <code>IsContainer</code> parameter for the System.Management.Automation.Provider.CmdletProvider.WriteLineObject * method.

ContainerCmdletProvider Base Class

The [System.Management.Automation.Provider.ContainerCmdletProvider](#) class defines a Windows PowerShell container provider that exposes a container, for data store items, to the user. Be aware that a Windows PowerShell container provider can be used only when there is one container (no nested containers) with items in it. If there are nested containers, then you must implement a Windows PowerShell navigation provider .

This class derives from the

[System.Management.Automation.Provider.ItemCmdletProvider](#) base class. The following table defines the cmdlets implemented by this class.

[+] Expand table

Cmdlet	Definition
Copy-Item	Copies items from one location to another. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.
Get-ChildItem	Retrieves the child items at the specified location, and streams them as objects.
New-Item	Creates new items at the specified location, and streams the resultant object.

Cmdlet	Definition
Remove-Item	Removes items from the specified location.
Rename-Item	Renames an item at the specified location. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.

NavigationCmdletProvider Base Class

The [System.Management.Automation.Provider.NavigationCmdletProvider](#) class defines a Windows PowerShell navigation provider that performs operations for items that use more than one container. This class derives from the [System.Management.Automation.Provider.ContainerCmdletProvider](#) base class. The following table lists the cmdlets exposed by this class.

[] [Expand table](#)

Cmdlet	Definition
Combine-Path	Combines two paths into a single path, using a provider-specific delimiter between paths. This cmdlet streams strings.
Move-Item	Moves items to the specified location. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.

A related cmdlet is the basic `Parse-Path` cmdlet furnished by Windows PowerShell. This cmdlet can be used to parse a Windows PowerShell path to support the `Parent` parameter. It streams the parent path string.

Select Provider Interfaces to Support

In addition to deriving from one of the Windows PowerShell base classes, your Windows PowerShell provider can support other functionality by deriving from one or more of the following provider interfaces. This section defines those interfaces and the cmdlets supported by each. It does not describe the parameters for the interface-supported cmdlets. Cmdlet parameter information is available online using the `Get-Command` and `Get-Help` cmdlets.

IContentCmdletProvider

The [System.Management.Automation.Provider.IContentCmdletProvider](#) interface defines a content provider that performs operations on the content of a data item. The following table lists the cmdlets exposed by this interface.

[+] [Expand table](#)

Cmdlet	Definition
Add-Content	Appends the indicated value lengths to the contents of the specified item. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.
Clear-Content	Sets the content of the specified item to the "clear" value. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.
Get-Content	Retrieves the contents of the specified items and streams the resultant objects.
Set-Content	Replaces the existing content for the specified items. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.

IPropertyCmdletProvider

The [System.Management.Automation.Provider.IPropertyCmdletProvider](#) interface defines a property Windows PowerShell provider that performs operations on the properties of items in the data store. The following table lists the cmdlets exposed by this interface.

ⓘ Note

The `Path` parameter on these cmdlets indicates a path to an item instead of identifying a property.

[+] [Expand table](#)

Cmdlet	Definition
Clear-ItemProperty	Sets properties of the specified items to the "clear" value. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.
Get-ItemProperty	Retrieves properties from the specified items and streams the resultant objects.

Cmdlet	Definition
Set- ItemProperty	Sets properties of the specified items with the indicated values. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.

IDynamicPropertyCmdletProvider

The [System.Management.Automation.Provider.IDynamicPropertyCmdletProvider](#) interface, derived from [System.Management.Automation.Provider.IPropertyCmdletProvider](#), defines a provider that specifies dynamic parameters for its supported cmdlets. This type of provider handles operations for which properties can be defined at run time, for example, a new property operation. Such operations are not possible on items having statically defined properties. The following table lists the cmdlets exposed by this interface.

[+] [Expand table](#)

Cmdlet	Definition
Copy- ItemProperty	Copies a property from the specified item to another item. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.
Move- ItemProperty	Moves a property from the specified item to another item. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.
New- ItemProperty	Creates a property on the specified items and streams the resultant objects.
Remove- ItemProperty	Removes a property for the specified items.
Rename- ItemProperty	Renames a property of the specified items. This cmdlet does not pass an output object through the pipeline unless its <code>PassThru</code> parameter is specified.

ISecurityDescriptorCmdletProvider

The [System.Management.Automation.Provider.ISecurityDescriptorCmdletProvider](#) interface adds security descriptor functionality to a provider. This interface allows the user to get and set security descriptor information for an item in the data store. The following table lists the cmdlets exposed by this interface.

Cmdlet	Definition
Get-Acl	Retrieves the information contained in an access control list (ACL), which is part of a security descriptor used to guard operating system resources, for example, a file or an object.
Set-Acl	Sets the information for an ACL. It is in the form of an instance of System.Security.AccessControl.ObjectSecurity on the item(s) designated for the specified path. This cmdlet can set information about files, keys, and subkeys in the registry, or any other provider item, if the Windows PowerShell provider supports the setting of security information.

See Also

[Creating Windows PowerShell Providers](#)

[How Windows PowerShell Works](#)

[Windows PowerShell SDK](#)

Creating a Basic Windows PowerShell Provider

Article • 09/17/2021

This topic is the starting point for learning how to create a Windows PowerShell provider. The basic provider described here provides methods for starting and stopping the provider, and although this provider does not provide a means to access a data store or to get or set the data in the data store, it does provide the basic functionality that is required by all providers.

As mentioned previously, the basic provider described here implements methods for starting and stopping the provider. The Windows PowerShell runtime calls these methods to initialize and uninitialized the provider.

ⓘ Note

You can find a sample of this provider in the `AccessDBSampleProvider01.cs` file provided by Windows PowerShell.

Defining the Windows PowerShell Provider Class

The first step in creating a Windows PowerShell provider is to define its .NET class. This basic provider defines a class called `AccessDBProvider` that derives from the [System.Management.Automation.Provider.CmdletProvider](#) base class.

It is recommended that you place your provider classes in a `Providers` namespace of your API namespace, for example, `xxx.PowerShell.Providers`. This provider uses the `Microsoft.Samples.PowerShell.Provider` namespace, in which all Windows PowerShell provider samples run.

ⓘ Note

The class for a Windows PowerShell provider must be explicitly marked as public. Classes not marked as public will default to internal and will not be found by the Windows PowerShell runtime.

Here is the class definition for this basic provider:

C#

```
[CmdletProvider("AccessDB", ProviderCapabilities.None)]
public class AccessDBProvider : CmdletProvider
```

Right before the class definition, you must declare the [System.Management.Automation.Provider.CmdletProviderAttribute](#) attribute, with the syntax [CmdletProvider()].

You can set attribute keywords to further declare the class if necessary. Notice that the [System.Management.Automation.Provider.CmdletProviderAttribute](#) attribute declared here includes two parameters. The first attribute parameter specifies the default-friendly name for the provider, which the user can modify later. The second parameter specifies the Windows PowerShell-defined capabilities that the provider exposes to the Windows PowerShell runtime during command processing. The possible values for the provider capabilities are defined by the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. Because this is a base provider, it supports no capabilities.

ⓘ Note

The fully qualified name of the Windows PowerShell provider includes the assembly name and other attributes determined by Windows PowerShell upon registration of the provider.

Defining Provider-Specific State Information

The [System.Management.Automation.Provider.CmdletProvider](#) base class and all derived classes are considered stateless because the Windows PowerShell runtime creates provider instances only as required. Therefore, if your provider requires full control and state maintenance for provider-specific data, it must derive a class from the [System.Management.Automation.ProviderInfo](#) class. Your derived class should define the members necessary to maintain the state so that the provider-specific data can be accessed when the Windows PowerShell runtime calls the [System.Management.Automation.Provider.CmdletProvider.Start*](#) method to initialize the provider.

A Windows PowerShell provider can also maintain connection-based state. For more information about maintaining connection state, see [Creating a PowerShell Drive Provider](#).

Initializing the Provider

To initialize the provider, the Windows PowerShell runtime calls the [System.Management.Automation.Provider.CmdletProvider.Start*](#) method when Windows PowerShell is started. For the most part, your provider can use the default implementation of this method, which simply returns the [System.Management.Automation.ProviderInfo](#) object that describes your provider. However, in the case where you want to add additional initialization information, you should implement your own [System.Management.Automation.Provider.CmdletProvider.Start*](#) method that returns a modified version of the [System.Management.Automation.ProviderInfo](#) object that is passed to your provider. In general, this method should return the provided [System.Management.Automation.ProviderInfo](#) object passed to it or a modified [System.Management.Automation.ProviderInfo](#) object that contains other initialization information.

This basic provider does not override this method. However, the following code shows the default implementation of this method:

The provider can maintain the state of provider-specific information as described in [Defining Provider-specific Data State](#). In this case, your implementation must override the [System.Management.Automation.Provider.CmdletProvider.Start*](#) method to return an instance of the derived class.

Start Dynamic Parameters

Your provider implementation of the [System.Management.Automation.Provider.CmdletProvider.Start*](#) method might require additional parameters. In this case, the provider should override the [System.Management.Automation.Provider.CmdletProvider.StartDynamicParameters*](#) method and return an object that has properties and fields with parsing attributes similar to a cmdlet class or a [System.Management.Automation.RuntimeDefinedParameterDictionary](#) object.

This basic provider does not override this method. However, the following code shows the default implementation of this method:

Uninitializing the Provider

To free resources that the Windows PowerShell provider uses, your provider should implement its own [System.Management.Automation.Provider.CmdletProvider.Stop*](#)

method. This method is called by the Windows PowerShell runtime to uninitialized the provider at the close of a session.

This basic provider does not override this method. However, the following code shows the default implementation of this method:

Code Sample

For complete sample code, see [AccessDbProviderSample01 Code Sample](#).

Testing the Windows PowerShell Provider

Once your Windows PowerShell provider has been registered with Windows PowerShell, you can test it by running the supported cmdlets on the command line. For this basic provider, run the new shell and use the `Get-PSProvider` cmdlet to retrieve the list of providers and ensure that the AccessDb provider is present.

```
PowerShell
Get-PSProvider
```

The following output appears:

```
Output
Name          Capabilities      Drives
----          -----
AccessDb      None
Alias         ShouldProcess
Environment   ShouldProcess
FileSystem    Filter, ShouldProcess
Function      ShouldProcess
Registry      ShouldProcess
```

See Also

[Creating Windows PowerShell Providers](#)

[Designing Your Windows PowerShell Provider](#)

Creating a Windows PowerShell Drive Provider

Article • 03/24/2025

This topic describes how to create a Windows PowerShell drive provider that provides a way to access a data store through a Windows PowerShell drive. This type of provider is also referred to as Windows PowerShell drive providers. The Windows PowerShell drives used by the provider provide the means to connect to the data store.

The Windows PowerShell drive provider described here provides access to a Microsoft Access database. For this provider, the Windows PowerShell drive represents the database (it is possible to add any number of drives to a drive provider), the top-level containers of the drive represent the tables in the database, and the items of the containers represent the rows in the tables.

Defining the Windows PowerShell Provider Class

Your drive provider must define a .NET class that derives from the [System.Management.Automation.Provider.DriveCmdletProvider](#) base class. Here is the class definition for this drive provider:

```
C#
```

```
[CmdletProvider("AccessDB", ProviderCapabilities.None)]
public class AccessDBProvider : DriveCmdletProvider
```

Notice that in this example, the [System.Management.Automation.Provider.CmdletProviderAttribute](#) attribute specifies a user-friendly name for the provider and the Windows PowerShell specific capabilities that the provider exposes to the Windows PowerShell runtime during command processing. The possible values for the provider capabilities are defined by the [System.Management.Automation.ProviderCapabilities](#) enumeration. This drive provider does not support any of these capabilities.

Defining Base Functionality

As described in [Design Your Windows PowerShell Provider](#), the [System.Management.Automation.Provider.DriveCmdletProvider](#) class derives from the

`System.Management.Automation.Provider.CmdletProvider` base class that defines the methods needed for initializing and uninitialized the provider. To implement functionality for adding session-specific initialization information and for releasing resources that are used by the provider, see [Creating a Basic Windows PowerShell Provider](#). However, most providers (including the provider described here) can use the default implementation of this functionality that is provided by Windows PowerShell.

Creating Drive State Information

All Windows PowerShell providers are considered stateless, which means that your drive provider needs to create any state information that is needed by the Windows PowerShell runtime when it calls your provider.

For this drive provider, state information includes the connection to the database that is kept as part of the drive information. Here is code that shows how this information is stored in the `System.Management.Automation.PSDriveinfo` object that describes the drive:

C#

```
internal class AccessDBPSDriveInfo : PSDriveInfo
{
    private OdbcConnection connection;

    /// <summary>
    /// ODBC connection information.
    /// </summary>
    public OdbcConnection Connection
    {
        get { return connection; }
        set { connection = value; }
    }

    /// <summary>
    /// Constructor that takes one argument
    /// </summary>
    /// <param name="driveInfo">Drive provided by this provider</param>
    public AccessDBPSDriveInfo(PSDriveInfo driveInfo)
        : base(driveInfo)
    { }

} // class AccessDBPSDriveInfo
```

Creating a Drive

To allow the Windows PowerShell runtime to create a drive, the drive provider must implement the [System.Management.Automation.Provider.DriveCmdletProvider.NewDrive*](#) method. The following code shows the implementation of the [System.Management.Automation.Provider.DriveCmdletProvider.NewDrive*](#) method for this drive provider:

C#

```
protected override PSDriveInfo NewDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            null)
        );
        return null;
    }

    // check if drive root is not null or empty
    // and if its an existing file
    if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root)
== false))
    {
        WriteError(new ErrorRecord(
            new ArgumentException("drive.Root"),
            "NoRoot",
            ErrorCategory.InvalidArgument,
            drive)
        );
        return null;
    }

    // create a new drive and create an ODBC connection to the new
    // drive
    AccessDBPSDriveInfo accessDBPSDriveInfo = new
    AccessDBPSDriveInfo(drive);

    OdbcConnectionStringBuilder builder = new
    OdbcConnectionStringBuilder();

    builder.Driver = "Microsoft Access Driver (*.mdb)";
    builder.Add("DBQ", drive.Root);

    OdbcConnection conn = new
    OdbcConnection(builder.ConnectionString);
```

```
conn.Open();
accessDBPSDriveInfo.Connection = conn;

return accessDBPSDriveInfo;
} // NewDrive
```

Your override of this method should do the following:

- Verify that the [System.Management.Automation.PSDriveinfo.Root*](#) member exists and that a connection to the data store can be made.
- Create a drive and populate the connection member, in support of the [New-PSDrive](#) cmdlet.
- Validate the [System.Management.Automation.PSDriveinfo](#) object for the proposed drive.
- Modify the [System.Management.Automation.PSDriveinfo](#) object that describes the drive with any required performance or reliability information, or provide extra data for callers using the drive.
- Handle failures using the [System.Management.Automation.Provider.CmdletProvider.WriteError](#) method and then return `null`.

This method returns either the drive information that was passed to the method or a provider-specific version of it.

Attaching Dynamic Parameters to NewDrive

The [New-PSDrive](#) cmdlet supported by your drive provider might require additional parameters. To attach these dynamic parameters to the cmdlet, the provider implements the

[System.Management.Automation.Provider.DriveCmdletProvider.NewDriveDynamicParameters*](#) method. This method returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a [System.Management.Automation.RuntimeDefinedParameterDictionary](#) object.

This drive provider does not override this method. However, the following code shows the default implementation of this method:

Removing a Drive

To close the database connection, the drive provider must implement the [System.Management.Automation.Provider.DriveCmdletProvider.RemoveDrive*](#) method. This method closes the connection to the drive after cleaning up any provider-specific information.

The following code shows the implementation of the [System.Management.Automation.Provider.DriveCmdletProvider.RemoveDrive*](#) method for this drive provider:

C#

```
protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            drive)
    );
    return null;
}

// close ODBC connection to the drive
AccessDBPSDriveInfo accessDBPSDriveInfo = drive as AccessDBPSDriveInfo;

if (accessDBPSDriveInfo == null)
{
    return null;
}
accessDBPSDriveInfo.Connection.Close();

return accessDBPSDriveInfo;
} // RemoveDrive
```

If the drive can be removed, the method should return the information passed to the method through the `drive` parameter. If the drive cannot be removed, the method should write an exception and then return `null`. If your provider does not override this method, the default implementation of this method just returns the drive information passed as input.

Initializing Default Drives

Your drive provider implements the [System.Management.Automation.Provider.DriveCmdletProvider.InitializeDefaultDrives*](#) method to mount drives. For example, the Active Directory provider might mount a drive for the default naming context if the computer is joined to a domain.

This method returns a collection of drive information about the initialized drives, or an empty collection. The call to this method is made after the Windows PowerShell runtime calls the [System.Management.Automation.Provider.CmdletProvider.Start*](#) method to initialize the provider.

This drive provider does not override the [System.Management.Automation.Provider.DriveCmdletProvider.InitializeDefaultDrives*](#) method. However, the following code shows the default implementation, which returns an empty drive collection:

Things to Remember About Implementing InitializeDefaultDrives

All drive providers should mount a root drive to help the user with discoverability. The root drive might list locations that serve as roots for other mounted drives. For example, the Active Directory provider might create a drive that lists the naming contexts found in the `namingContext` attributes on the root Distributed System Environment (DSE). This helps users discover mount points for other drives.

Code Sample

For complete sample code, see [AccessDbProviderSample02 Code Sample](#).

Testing the Windows PowerShell Drive Provider

When your Windows PowerShell provider has been registered with Windows PowerShell, you can test it by running the supported cmdlets on the command line, including any cmdlets made available by derivation. Let's test the sample drive provider.

1. Run the `Get-PSPProvider` cmdlet to retrieve the list of providers to ensure that the AccessDB drive provider is present:

```
PS> Get-PSPProvider
```

The following output appears:

```
Output
```

Name	Capabilities	Drives
AccessDB	None	{}
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess	{C, Z}
Function	ShouldProcess	{function}
Registry	ShouldProcess	{HKLM, HKCU}

2. Ensure that a database server name (DSN) exists for the database by accessing the **Data Sources** portion of the **Administrative Tools** for the operating system. In the **User DSN** table, double-click **MS Access Database** and add the drive path

C:\ps\northwind.mdb.

3. Create a new drive using the sample drive provider:

PowerShell

```
New-PSDrive -Name mydb -Root C:\ps\northwind.mdb -PSPrinter AccessDb`
```

The following output appears:

Output

Name	Provider	Root	CurrentLocation
mydb	AccessDB	C:\ps\northwind.mdb	

4. Validate the connection. Because the connection is defined as a member of the drive, you can check it using the Get-PDDrive cmdlet.

(!) Note

The user cannot yet interact with the provider as a drive, as the provider needs container functionality for that interaction. For more information, see [Creating a Windows PowerShell Container Provider](#).

PS> (Get-PSDrive mydb).Connection

The following output appears:

Output

```
ConnectionString : Driver={Microsoft Access Driver (*.mdb)};DBQ=C:\ps\northwind.mdb
ConnectionTimeout : 15
Database        : C:\ps\northwind
DataSource       : ACCESS
ServerVersion    : 04.00.0000
Driver           : odbcjt32.dll
State            : Open
Site             :
Container        :
```

5. Remove the drive and exit the shell:

```
PowerShell
```

```
PS> Remove-PSDrive mydb
PS> exit
```

See Also

[Creating Windows PowerShell Providers](#)

[Design Your Windows PowerShell Provider](#)

[Creating a Basic Windows PowerShell Provider](#)

Creating a Windows PowerShell item provider

Article • 05/12/2022

This topic describes how to create a Windows PowerShell provider that can manipulate the data in a data store. In this topic, the elements of data in the store are referred to as the "items" of the data store. As a consequence, a provider that can manipulate the data in the store is referred to as a Windows PowerShell item provider.

ⓘ Note

You can download the C# source file (`AccessDBSampleProvider03.cs`) for this provider using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the `PowerShell Samples` directory. For more information about other Windows PowerShell provider implementations, see [Designing Your Windows PowerShell Provider](#).

The Windows PowerShell item provider described in this topic gets items of data from an Access database. In this case, an "item" is either a table in the Access database or a row in a table.

Defining the Windows PowerShell item provider class

A Windows PowerShell item provider must define a .NET class that derives from the [System.Management.Automation.Provider.ItemCmdletProvider](#) base class. The following is the class definition for the item provider described in this section.

```
C#
```

```
[CmdletProvider("AccessDB", ProviderCapabilities.None)]  
public class AccessDBProvider : ItemCmdletProvider
```

Note that in this class definition, the [System.Management.Automation.Provider.CmdletProviderAttribute](#) attribute includes two parameters. The first parameter specifies a user-friendly name for the provider that

is used by Windows PowerShell. The second parameter specifies the Windows PowerShell specific capabilities that the provider exposes to the Windows PowerShell runtime during command processing. For this provider, there are no added Windows PowerShell specific capabilities.

Defining base functionality

As described in [Design Your Windows PowerShell Provider](#), the `System.Management.Automation.Provider.DriveCmdletProvider` class derives from several other classes that provided different provider functionality. A Windows PowerShell item provider, therefore, must define all of the functionality provided by those classes.

For more information about how to implement functionality for adding session-specific initialization information and for releasing resources used by the provider, see [Creating a Basic Windows PowerShell Provider](#). However, most providers, including the provider described here, can use the default implementation of this functionality that is provided by Windows PowerShell.

Before the Windows PowerShell item provider can manipulate the items in the store, it must implement the methods of the `System.Management.Automation.Provider.DriveCmdletProvider` base class to access to the data store. For more information about implementing this class, see [Creating a Windows PowerShell Drive Provider](#).

Checking for path validity

When looking for an item of data, the Windows PowerShell runtime furnishes a Windows PowerShell path to the provider, as defined in the "PSPPath Concepts" section of [How Windows PowerShell Works](#). A Windows PowerShell item provider must verify the syntactic and semantic validity of any path passed to it by implementing the `System.Management.Automation.Provider.ItemCmdletProvider.IsValidPath` method. This method returns `true` if the path is valid, and `false` otherwise. Be aware that the implementation of this method should not verify the existence of the item at the path, but only that the path is syntactically and semantically correct.

Here is the implementation of the `System.Management.Automation.Provider.ItemCmdletProvider.IsValidPath` method for this provider. Note that this implementation calls a `NormalizePath` helper method to convert all separators in the path to a uniform one.

C#

```
protected override bool IsValidPath(string path)
{
    bool result = true;

    // check if the path is null or empty
    if (String.IsNullOrEmpty(path))
    {
        result = false;
    }

    // convert all separators in the path to a uniform one
    path = NormalizePath(path);

    // split the path into individual chunks
    string[] pathChunks = path.Split(pathSeparator.ToCharArray());

    foreach (string pathChunk in pathChunks)
    {
        if (pathChunk.Length == 0)
        {
            result = false;
        }
    }
    return result;
} // IsValidPath
```

Determining if an item exists

After verifying the path, the Windows PowerShell runtime must determine if an item of data exists at that path. To support this type of query, the Windows PowerShell item provider implements the

[System.Management.Automation.Provider.ItemCmdletProvider.ItemExists](#) method. This method returns `true` if an item is found at the specified path and `false` (default) otherwise.

Here is the implementation of the

[System.Management.Automation.Provider.ItemCmdletProvider.ItemExists](#) method for this provider. Note that this method calls the `PathIsDrive`, `ChunkPath`, and `GetTable` helper methods, and uses a provider defined `DatabaseTableInfo` object.

C#

```
protected override bool ItemExists(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
```

```

    {
        return true;
    }

    // Obtain type, table name and row number from path
    string tableName;
    int rowCount;

    PathType type = GetNamesFromPath(path, out tableName, out rowCount);

    DatabaseTableInfo table = GetTable(tableName);

    if (type == PathType.Table)
    {
        // if specified path represents a table then DatabaseTableInfo
        // object for the same should exist
        if (table != null)
        {
            return true;
        }
    }
    else if (type == PathType.Row)
    {
        // if specified path represents a row then DatabaseTableInfo should
        // exist for the table and then specified row number must be within
        // the maximum row count in the table
        if (table != null && rowCount < table.RowCount)
        {
            return true;
        }
    }

    return false;
}

} // ItemExists

```

Things to remember about implementing ItemExists

The following conditions may apply to your implementation of [System.Management.Automation.Provider.ItemCmdletProvider.ItemExists](#):

- When defining the provider class, a Windows PowerShell item provider might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of the [System.Management.Automation.Provider.ItemCmdletProvider.ItemExists](#) method must ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the

`System.Management.Automation.Provider.CmdletProvider.Exclude` and
`System.Management.Automation.Provider.CmdletProvider.Include` properties.

- The implementation of this method should handle any form of access to the item that might make the item visible to the user. For example, if a user has write access to a file through the `FileSystem` provider (supplied by Windows PowerShell), but not read access, the file still exists and `System.Management.Automation.Provider.ItemCmdletProvider.ItemExists` returns `true`. Your implementation might require checking a parent item to see if the child item can be enumerated.

Attaching dynamic parameters to the `Test-Path` cmdlet

Sometimes the `Test-Path` cmdlet calls

`System.Management.Automation.Provider.ItemCmdletProvider.ItemExists` requires additional parameters that are specified dynamically at runtime. To provide these dynamic parameters the Windows PowerShell item provider must implement the `System.Management.Automation.Provider.ItemCmdletProvider.ItemExistsDynamicParameters` method. This method retrieves the dynamic parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a `System.Management.Automation.RuntimeDefinedParameterDictionary` object. The Windows PowerShell runtime uses the returned object to add the parameters to the `Test-Path` cmdlet.

This Windows PowerShell item provider does not implement this method. However, the following code is the default implementation of this method.

Retrieving an item

To retrieve an item, the Windows PowerShell item provider must override `System.Management.Automation.Provider.ItemCmdletProvider.GetItem` method to support calls from the `Get-Item` cmdlet. This method writes the item using the `System.Management.Automation.Provider.CmdletProvider.WriteLineObject` method.

Here is the implementation of the

`System.Management.Automation.Provider.ItemCmdletProvider.GetItem` method for this provider. Note that this method uses the `GetTable` and `GetRow` helper methods to retrieve items that are either tables in the Access database or rows in a data table.

C#

```
protected override void GetItem(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        WriteItemObject(this.PSDriveInfo, path, true);
        return;
    } // if (PathIsDrive...

    // Get table name and row information from the path and do
    // necessary actions
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out rowNumber);

    if (type == PathType.Table)
    {
        DatabaseTableInfo table = GetTable(tableName);
        WriteItemObject(table, path, true);
    }
    else if (type == PathType.Row)
    {
        DatabaseRowInfo row = GetRow(tableName, rowNumber);
        WriteItemObject(row, path, false);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }

} // GetItem
```

Things to remember about implementing GetItem

The following conditions may apply to an implementation of [System.Management.Automation.Provider.ItemCmdletProvider.GetItem](#):

- When defining the provider class, a Windows PowerShell item provider might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of [System.Management.Automation.Provider.ItemCmdletProvider.GetItem](#) must ensure that the path passed to the method meets those requirements. To do this, the method should access the appropriate property, for example, the

`System.Management.Automation.Provider.CmdletProvider.Exclude` and `System.Management.Automation.Provider.CmdletProvider.Include` properties.

- By default, overrides of this method should not retrieve objects that are generally hidden from the user unless the `System.Management.Automation.Provider.CmdletProvider.Force` property is set to `true`. For example, the `System.Management.Automation.Provider.ItemCmdletProvider.GetItem` method for the `FileSystem` provider checks the `System.Management.Automation.Provider.CmdletProvider.Force` property before it attempts to call `System.Management.Automation.Provider.CmdletProvider.WriteLineObject` for hidden or system files.

Attaching dynamic parameters to the Get-Item cmdlet

Sometimes the `Get-Item` cmdlet requires additional parameters that are specified dynamically at runtime. To provide these dynamic parameters the Windows PowerShell item provider must implement the `System.Management.Automation.Provider.ItemCmdletProvider.GetItemDynamicParameters` method. This method retrieves the dynamic parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a `System.Management.Automation.RuntimeDefinedParameterDictionary` object. The Windows PowerShell runtime uses the returned object to add the parameters to the `Get-Item` cmdlet.

This provider does not implement this method. However, the following code is the default implementation of this method.

Setting an item

To set an item, the Windows PowerShell item provider must override the `System.Management.Automation.Provider.ItemCmdletProvider.SetItem` method to support calls from the `Set-Item` cmdlet. This method sets the value of the item at the specified path.

This provider does not provide an override for the `System.Management.Automation.Provider.ItemCmdletProvider.SetItem` method.

However, the following is the default implementation of this method.

Things to remember about implementing SetItem

The following conditions may apply to your implementation of [System.Management.Automation.Provider.ItemCmdletProvider.SetItem](#):

- When defining the provider class, a Windows PowerShell item provider might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of [System.Management.Automation.Provider.ItemCmdletProvider.SetItem](#) must ensure that the path passed to the method meets those requirements. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude](#) and [System.Management.Automation.Provider.CmdletProvider.Include](#) properties.
- By default, overrides of this method should not set or write objects that are hidden from the user unless the [System.Management.Automation.Provider.CmdletProvider.Force](#) property is set to `true`. An error should be sent to the [System.Management.Automation.Provider.CmdletProvider.WriteError](#) method if the path represents a hidden item and [System.Management.Automation.Provider.CmdletProvider.Force](#) is set to `false`.
- Your implementation of the [System.Management.Automation.Provider.ItemCmdletProvider.SetItem](#) method should call [System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) and verify its return value before making any changes to the data store. This method is used to confirm execution of an operation when a change is made to the data store, for example, deleting files. The [System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) method sends the name of the resource to be changed to the user, with the Windows PowerShell runtime taking into account any command-line settings or preference variables in determining what should be displayed.

After the call to

[System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) returns `true`, the [System.Management.Automation.Provider.ItemCmdletProvider.SetItem](#) method should call the [System.Management.Automation.Provider.CmdletProvider.ShouldContinue](#)

method. This method sends a message to the user to allow feedback to verify if the operation should be continued. The call to [System.Management.Automation.Provider.CmdletProvider.ShouldContinue](#) allows an additional check for potentially dangerous system modifications.

Retrieving dynamic parameters for SetItem

Sometimes the `Set-Item` cmdlet requires additional parameters that are specified dynamically at runtime. To provide these dynamic parameters the Windows PowerShell item provider must implement the

[System.Management.Automation.Provider.ItemCmdletProvider.SetItemDynamicParameters](#) method. This method retrieves the dynamic parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a

[System.Management.Automation.RuntimeDefinedParameterDictionary](#) object. The Windows PowerShell runtime uses the returned object to add the parameters to the `Set-Item` cmdlet.

This provider does not implement this method. However, the following code is the default implementation of this method.

Clearing an item

To clear an item, the Windows PowerShell item provider implements the [System.Management.Automation.Provider.ItemCmdletProvider.ClearItem](#) method to support calls from the `Clear-Item` cmdlet. This method erases the data item at the specified path.

This provider does not implement this method. However, the following code is the default implementation of this method.

Things to remember about implementing ClearItem

The following conditions may apply to an implementation of [System.Management.Automation.Provider.ItemCmdletProvider.ClearItem](#):

- When defining the provider class, a Windows PowerShell item provider might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of [System.Management.Automation.Provider.ItemCmdletProvider.ClearItem](#) must

ensure that the path passed to the method meets those requirements. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude](#) and [System.Management.Automation.Provider.CmdletProvider.Include](#) properties.

- By default, overrides of this method should not set or write objects that are hidden from the user unless the [System.Management.Automation.Provider.CmdletProvider.Force](#) property is set to `true`. An error should be sent to the [System.Management.Automation.Provider.CmdletProvider.WriteError](#) method if the path represents an item that is hidden from the user and [System.Management.Automation.Provider.CmdletProvider.Force](#) is set to `false`.
- Your implementation of the [System.Management.Automation.Provider.ItemCmdletProvider.SetItem](#) method should call [System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) and verify its return value before making any changes to the data store. This method is used to confirm execution of an operation when a change is made to the data store, for example, deleting files. The [System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) method sends the name of the resource to be changed to the user, with the Windows PowerShell runtime and handle any command-line settings or preference variables in determining what should be displayed.

After the call to

[System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) returns `true`, the [System.Management.Automation.Provider.ItemCmdletProvider.SetItem](#) method should call the [System.Management.Automation.Provider.CmdletProvider.ShouldContinue](#) method. This method sends a message to the user to allow feedback to verify if the operation should be continued. The call to [System.Management.Automation.Provider.CmdletProvider.ShouldContinue](#) allows an additional check for potentially dangerous system modifications.

Retrieve dynamic parameters for ClearItem

Sometimes the `Clear-Item` cmdlet requires additional parameters that are specified dynamically at runtime. To provide these dynamic parameters the Windows PowerShell item provider must implement the [System.Management.Automation.Provider.ItemCmdletProvider.ClearItemDynamicParam](#)

`eters` method. This method retrieves the dynamic parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a `System.Management.Automation.RuntimeDefinedParameterDictionary` object. The Windows PowerShell runtime uses the returned object to add the parameters to the `Clear-Item` cmdlet.

This item provider does not implement this method. However, the following code is the default implementation of this method.

Performing a default action for an item

A Windows PowerShell item provider can implement the `System.Management.Automation.Provider.ItemCmdletProvider.InvokeDefaultAction` method to support calls from the `Invoke-Item` cmdlet, which allows the provider to perform a default action for the item at the specified path. For example, the FileSystem provider might use this method to call `ShellExecute` for a specific item.

This provider does not implement this method. However, the following code is the default implementation of this method.

Things to remember about implementing `InvokeDefaultAction`

The following conditions may apply to an implementation of `System.Management.Automation.Provider.ItemCmdletProvider.InvokeDefaultAction`:

- When defining the provider class, a Windows PowerShell item provider might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the `System.Management.Automation.Provider.ProviderCapabilities` enumeration. In these cases, the implementation of `System.Management.Automation.Provider.ItemCmdletProvider.InvokeDefaultAction` must ensure that the path passed to the method meets those requirements. To do this, the method should access the appropriate property, for example, the `System.Management.Automation.Provider.CmdletProvider.Exclude` and `System.Management.Automation.Provider.CmdletProvider.Include` properties.
- By default, overrides of this method should not set or write objects hidden from the user unless the `System.Management.Automation.Provider.CmdletProvider.Force` property is set to `true`. An error should be sent to the `System.Management.Automation.Provider.CmdletProvider.WriteError` method if

the path represents an item that is hidden from the user and [System.Management.Automation.Provider.CmdletProvider.Force](#) is set to `false`.

Retrieve dynamic parameters for InvokeDefaultAction

Sometimes the [Invoke-Item](#) cmdlet requires additional parameters that are specified dynamically at runtime. To provide these dynamic parameters the Windows PowerShell item provider must implement the

[System.Management.Automation.Provider.ItemCmdletProvider.InvokeDefaultActionDynamicParameters](#) method. This method retrieves the dynamic parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a [System.Management.Automation.RuntimeDefinedParameterDictionary](#) object. The Windows PowerShell runtime uses the returned object to add the dynamic parameters to the [Invoke-Item](#) cmdlet.

This item provider does not implement this method. However, the following code is the default implementation of this method.

Implementing helper methods and classes

This item provider implements several helper methods and classes that are used by the public override methods defined by Windows PowerShell. The code for these helper methods and classes are shown in the [Code Sample](#) section.

NormalizePath method

This item provider implements a **NormalizePath** helper method to ensure that the path has a consistent format. The format specified uses a backslash (\) as a separator.

PathIsDrive method

This item provider implements a **PathIsDrive** helper method to determine if the specified path is actually the drive name.

ChunkPath method

This item provider implements a **ChunkPath** helper method that breaks up the specified path so that the provider can identify its individual elements. It returns an array composed of the path elements.

GetTable method

This item provider implements the **GetTables** helper method that returns a **DatabaseTableInfo** object that represents information about the table specified in the call.

GetRow method

The [System.Management.Automation.Provider.ItemCmdletProvider.GetItem](#) method of this item provider calls the **GetRows** helper method. This helper method retrieves a **DatabaseRowInfo** object that represents information about the specified row in the table.

DatabaseTableInfo class

This item provider defines a **DatabaseTableInfo** class that represents a collection of information in a data table in the database. This class is similar to the [System.IO.DirectoryInfo](#) class.

The sample item provider defines a **DatabaseTableInfo.GetTables** method that returns a collection of table information objects defining the tables in the database. This method includes a try/catch block to ensure that any database error shows up as a row with zero entries.

DatabaseRowInfo class

This item provider defines the **DatabaseRowInfo** helper class that represents a row in a table of the database. This class is similar to the [System.IO.FileInfo](#) class.

The sample provider defines a **DatabaseRowInfo.GetRows** method to return a collection of row information objects for the specified table. This method includes a try/catch block to trap exceptions. Any errors will result in no row information.

Code sample

For complete sample code, see [AccessDbProviderSample03 Code Sample](#).

Defining object types and formatting

When writing a provider, it may be necessary to add members to existing objects or define new objects. When finished, create a Types file that Windows PowerShell can use to identify the members of the object and a Format file that defines how the object is displayed. For more information, see [Extending Object Types and Formatting](#).

Building the Windows PowerShell provider

See [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Windows PowerShell provider

When this Windows PowerShell item provider is registered with Windows PowerShell, you can only test the basic and drive functionality of the provider. To test the manipulation of items, you must also implement container functionality described in [Implementing a Container Windows PowerShell Provider](#).

See also

- [Windows PowerShell SDK](#)
- [Windows PowerShell Programmer's Guide](#)
- [Creating Windows PowerShell Providers](#)
- [Designing Your Windows PowerShell provider](#)
- [Extending Object Types and Formatting](#)
- [How Windows PowerShell Works](#)
- [Creating a Container Windows PowerShell provider](#)
- [Creating a Drive Windows PowerShell provider](#)
- [How to Register Cmdlets, Providers, and Host Applications](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

more information, see [our contributor guide](#).

Creating a Windows PowerShell Container Provider

Article • 03/24/2025

This topic describes how to create a Windows PowerShell provider that can work on multi-layer data stores. For this type of data store, the top level of the store contains the root items and each subsequent level is referred to as a node of child items. By allowing the user to work on these child nodes, a user can interact hierarchically through the data store.

Providers that can work on multi-level data stores are referred to as Windows PowerShell container providers. However, be aware that a Windows PowerShell container provider can be used only when there is one container (no nested containers) with items in it. If there are nested containers, then you must implement a Windows PowerShell navigation provider. For more information about implementing Windows PowerShell navigation provider, see [Creating a Windows PowerShell Navigation Provider](#).

ⓘ Note

You can download the C# source file (AccessDBSampleProvider04.cs) for this provider using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory. For more information about other Windows PowerShell provider implementations, see [Designing Your Windows PowerShell Provider](#).

The Windows PowerShell container provider described here defines the database as its single container, with the tables and rows of the database defined as items of the container.

✖ Caution

Be aware that this design assumes a database that has a field with the name ID, and that the type of the field is LongInteger.

Defining a Windows PowerShell Container Provider Class

A Windows PowerShell container provider must define a .NET class that derives from the [System.Management.Automation.Provider.ContainerCmdletProvider](#) base class. Here is the class definition for the Windows PowerShell container provider described in this section.

C#

```
[CmdletProvider("AccessDB", ProviderCapabilities.None)]
public class AccessDBProvider : ContainerCmdletProvider
```

Notice that in this class definition, the [System.Management.Automation.Provider.CmdletProviderAttribute](#) attribute includes two parameters. The first parameter specifies a user-friendly name for the provider that is used by Windows PowerShell. The second parameter specifies the Windows PowerShell specific capabilities that the provider exposes to the Windows PowerShell runtime during command processing. For this provider, there are no Windows PowerShell specific capabilities that are added.

Defining Base Functionality

As described in [Designing Your Windows PowerShell Provider](#), the [System.Management.Automation.Provider.ContainerCmdletProvider](#) class derives from several other classes that provided different provider functionality. A Windows PowerShell container provider, therefore, needs to define all of the functionality provided by those classes.

To implement functionality for adding session-specific initialization information and for releasing resources that are used by the provider, see [Creating a Basic Windows PowerShell Provider](#). However, most providers (including the provider described here) can use the default implementation of this functionality that is provided by Windows PowerShell.

To get access to the data store, the provider must implement the methods of the [System.Management.Automation.Provider.DriveCmdletProvider](#) base class. For more information about implementing these methods, see [Creating a Windows PowerShell Drive Provider](#).

To manipulate the items of a data store, such as getting, setting, and clearing items, the provider must implement the methods provided by the [System.Management.Automation.Provider.ItemCmdletProvider](#) base class. For more information about implementing these methods, see [Creating a Windows PowerShell Item Provider](#).

Retrieving Child Items

To retrieve a child item, the Windows PowerShell container provider must override the [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems*](#) method to support calls from the `Get-ChildItem` cmdlet. This method retrieves child items from the data store and writes them to the pipeline as objects. If the `recurse` parameter of the cmdlet is specified, the method retrieves all children regardless of what level they are at. If the `recurse` parameter is not specified, the method retrieves only a single level of children.

Here is the implementation of the [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems*](#) method for this provider. Notice that this method retrieves the child items in all database tables when the path indicates the Access database, and retrieves the child items from the rows of that table if the path indicates a data table.

C#

```
protected override void GetChildItems(string path, bool recurse)
{
    // If path represented is a drive then the children in the path are
    // tables. Hence all tables in the drive represented will have to be
    // returned
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table, path, true);

            // if the specified item exists and recurse has been set then
            // all child items within it have to be obtained as well
            if (ItemExists(path) && recurse)
            {
                GetChildItems(path + pathSeparator + table.Name, recurse);
            }
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get the table name, row number and type of path from the
        // path specified
```

```

        string tableName;
        int rowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Table)
        {
            // Obtain all the rows within the table
            foreach (DatabaseRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row, path + pathSeparator + row.RowNumber,
                    false);
            } // foreach (DatabaseRowInfo...
        }
        else if (type == PathType.Row)
        {
            // In this case the user has directly specified a row, hence
            // just give that particular row
            DatabaseRowInfo row = GetRow(tableName, rowNumber);
            WriteItemObject(row, path + pathSeparator + row.RowNumber,
                false);
        }
        else
        {
            // In this case, the path specified is not valid
            ThrowTerminatingInvalidOperationException(path);
        }
    } // else
} // GetChildItems

```

Things to Remember About Implementing GetChildItems

The following conditions may apply to your implementation of [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems*](#):

- When defining the provider class, a Windows PowerShell container provider might declare provider capabilities of ExpandWildcards, Filter, Include, or Exclude, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of the [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems](#) * method needs to ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude*](#) and [System.Management.Automation.Provider.CmdletProvider.Include*](#) properties.

- The implementation of this method should take into account any form of access to the item that might make the item visible to the user. For example, if a user has write access to a file through the FileSystem provider (supplied by Windows PowerShell), but not read access, the file still exists and [System.Management.Automation.Provider.ItemCmdletProvider.ItemExists*](#) returns `true`. Your implementation might require the checking of a parent item to see if the child can be enumerated.
- When writing multiple items, the [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems*](#) method can take some time. You can design your provider to write the items using the [System.Management.Automation.Provider.CmdletProvider.WriteLineObject*](#) method one at a time. Using this technique will present the items to the user in a stream.
- Your implementation of [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems*](#) is responsible for preventing infinite recursion when there are circular links, and the like. An appropriate terminating exception should be thrown to reflect such a condition.

Attaching Dynamic Parameters to the Get-ChildItem Cmdlet

Sometimes the `Get-ChildItem` cmdlet that calls

[System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems*](#) requires additional parameters that are specified dynamically at runtime. To provide these dynamic parameters, the Windows PowerShell container provider must implement the

[System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItemsDynamicParameters*](#) method. This method retrieves dynamic parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a

[System.Management.Automation.RuntimeDefinedParameterDictionary](#) object. The Windows PowerShell runtime uses the returned object to add the parameters to the `Get-ChildItem` cmdlet.

This Windows PowerShell container provider does not implement this method. However, the following code is the default implementation of this method.

Retrieving Child Item Names

To retrieve the names of child items, the Windows PowerShell container provider must override the

`System.Management.Automation.Provider.ContainerCmdletProvider.GetChildNames*` method to support calls from the `Get-ChildItem` cmdlet when its `Name` parameter is specified. This method retrieves the names of the child items for the specified path or child item names for all containers if the `returnAllContainers` parameter of the cmdlet is specified. A child name is the leaf portion of a path. For example, the child name for the path `C:\windows\system32\abc.dll` is "abc.dll". The child name for the directory `C:\windows\system32` is "system32".

Here is the implementation of the

`System.Management.Automation.Provider.ContainerCmdletProvider.GetChildNames*` method for this provider. Notice that the method retrieves table names if the specified path indicates the Access database (drive) and row numbers if the path indicates a table.

C#

```
protected override void GetChildNames(string path,
                                      ReturnContainers returnContainers)
{
    // If the path represented is a drive, then the child items are
    // tables. get the names of all the tables in the drive.
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table.Name, path, true);
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get type, table name and row number from path specified
        string tableName;
        int rowCount;

        PathType type = GetNamesFromPath(path, out tableName, out
rowCount);

        if (type == PathType.Table)
        {
            // Get all the rows in the table and then write out the
            // row numbers.
            foreach (DatabaseRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row.RowNumber, path, false);
            } // foreach (DatabaseRowInfo...
        }
    }
}
```

```

    else if (type == PathType.Row)
    {
        // In this case the user has directly specified a row, hence
        // just give that particular row
        DatabaseRowInfo row = GetRow(tableName, rowNumber);

        WriteItemObject(row.RowNumber, path, false);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }
} // else
} // GetChildNames

```

Things to Remember About Implementing GetChildNames

The following conditions may apply to your implementation of [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems*](#):

- When defining the provider class, a Windows PowerShell container provider might declare provider capabilities of ExpandWildcards, Filter, Include, or Exclude, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of the [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems](#) * method needs to ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude*](#) and [System.Management.Automation.Provider.CmdletProvider.Include*](#) properties.

① Note

An exception to this rule occurs when the `returnAllContainers` parameter of the cmdlet is specified. In this case, the method should retrieve any child name for a container, even if it does not match the values of the [System.Management.Automation.Provider.CmdletProvider.Filter*](#), [System.Management.Automation.Provider.CmdletProvider.Include*](#), or [System.Management.Automation.Provider.CmdletProvider.Exclude*](#) properties.

- By default, overrides of this method should not retrieve names of objects that are generally hidden from the user unless the

`System.Management.Automation.Provider.CmdletProvider.Force`* property is specified. If the specified path indicates a container, the `System.Management.Automation.Provider.CmdletProvider.Force`* property is not required.

- Your implementation of `System.Management.Automation.Provider.ContainerCmdletProvider.GetChildNamesDynamicParameters`* is responsible for preventing infinite recursion when there are circular links, and the like. An appropriate terminating exception should be thrown to reflect such a condition.

Attaching Dynamic Parameters to the Get-ChildItem Cmdlet (Name)

Sometimes the `Get-ChildItem` cmdlet (with the `Name` parameter) requires additional parameters that are specified dynamically at runtime. To provide these dynamic parameters, the Windows PowerShell container provider must implement the `System.Management.Automation.Provider.ContainerCmdletProvider.GetChildNamesDynamicParameters`* method. This method retrieves the dynamic parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a `System.Management.Automation.RuntimeDefinedParameterDictionary` object. The Windows PowerShell runtime uses the returned object to add the parameters to the `Get-ChildItem` cmdlet.

This provider does not implement this method. However, the following code is the default implementation of this method.

Renaming Items

To rename an item, a Windows PowerShell container provider must override the `System.Management.Automation.Provider.ContainerCmdletProvider.RenameItem`* method to support calls from the `Rename-Item` cmdlet. This method changes the name of the item at the specified path to the new name provided. The new name must always be relative to the parent item (container).

This provider does not override the `System.Management.Automation.Provider.ContainerCmdletProvider.RenameItem`* method. However, the following is the default implementation.

Things to Remember About Implementing RenameItem

The following conditions may apply to your implementation of [System.Management.Automation.Provider.ContainerCmdletProvider.RenameItem](#)*:

- When defining the provider class, a Windows PowerShell container provider might declare provider capabilities of ExpandWildcards, Filter, Include, or Exclude, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of the [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems](#) * method needs to ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude](#)* and [System.Management.Automation.Provider.CmdletProvider.Include](#)* properties.
- The [System.Management.Automation.Provider.ContainerCmdletProvider.RenameItem](#)* method is intended for the modification of the name of an item only, and not for move operations. Your implementation of the method should write an error if the `newName` parameter contains path separators, or might otherwise cause the item to change its parent location.
- By default, overrides of this method should not rename objects unless the [System.Management.Automation.Provider.CmdletProvider.Force](#)* property is specified. If the specified path indicates a container, the [System.Management.Automation.Provider.CmdletProvider.Force](#)* property is not required.
- Your implementation of the [System.Management.Automation.Provider.ContainerCmdletProvider.RenameItem](#)* method should call [System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) and check its return value before making any changes to the data store. This method is used to confirm execution of an operation when a change is made to system state, for example, renaming files.
[System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) sends the name of the resource to be changed to the user, with the Windows PowerShell runtime taking into account any command line settings or preference variables in determining what should be displayed.

After the call to

[System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) returns

`true`, the

[System.Management.Automation.Provider.ContainerCmdletProvider.RenameItem*](#) method should call the [System.Management.Automation.Provider.CmdletProvider.ShouldContinue](#) method. This method sends a message a confirmation message to the user to allow additional feedback to say if the operation should be continued. A provider should call [System.Management.Automation.Provider.CmdletProvider.ShouldContinue](#) as an additional check for potentially dangerous system modifications.

Attaching Dynamic Parameters to the Rename-Item Cmdlet

Sometimes the `Rename-Item` cmdlet requires additional parameters that are specified dynamically at runtime. To provide these dynamic parameters, Windows PowerShell container provider must implement the

[System.Management.Automation.Provider.ContainerCmdletProvider.RenameItemDynamicParameters*](#) method. This method retrieves the parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a [System.Management.Automation.RuntimeDefinedParameterDictionary](#) object. The Windows PowerShell runtime uses the returned object to add the parameters to the `Rename-Item` cmdlet.

This container provider does not implement this method. However, the following code is the default implementation of this method.

Creating New Items

To create new items, a container provider must implement the

[System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*](#) method to support calls from the `New-Item` cmdlet. This method creates a data item located at the specified path. The `type` parameter of the cmdlet contains the provider-defined type for the new item. For example, the FileSystem provider uses a `type` parameter with a value of "file" or "directory". The `newValue` parameter of the cmdlet specifies a provider-specific value for the new item.

Here is the implementation of the

[System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*](#) method for this provider.

C#

```
protected override void NewItem( string path, string type, object newItemValue )
{
    // Create the new item here after
    // performing necessary validations
    //
    // WriteItemObject(newItemValue, path, false);

    // Example
    //
    // if (ShouldProcess(path, "new item"))
    // {
    //     // Create a new item and then call WriteObject
    //     WriteObject(newItemValue, path, false);
    // }

} // NewItem
```

C#

```
{
    case 1:
    {
        string name = pathChunks[0];

        if (TableNameIsValid(name))
        {
            tableName = name;
            retVal = PathType.Table;
        }
    }
    break;

    case 2:
    {
        string name = pathChunks[0];
```

Things to Remember About Implementing NewItem

The following conditions may apply to your implementation of [System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*](#):

- The [System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*](#) method should perform a case-insensitive comparison of the string passed in the `type` parameter. It should also allow for least ambiguous matches. For example, for the types "file" and "directory", only the first letter is required to disambiguate. If

the `type` parameter indicates a type your provider cannot create, the `System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*` method should write an `ArgumentException` with a message indicating the types the provider can create.

- For the `newValue` parameter, the implementation of the `System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*` method is recommended to accept strings at a minimum. It should also accept the type of object that is retrieved by the `System.Management.Automation.Provider.ItemCmdletProvider.GetItem*` method for the same path. The `System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*` method can use the `System.Management.Automation.LanguagePrimitives.ConvertTo*` method to convert types to the desired type.
- Your implementation of the `System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*` method should call `System.Management.Automation.Provider.CmdletProvider.ShouldProcess` and check its return value before making any changes to the data store. After the call to `System.Management.Automation.Provider.CmdletProvider.ShouldProcess` returns true, the `System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*` method should call the `System.Management.Automation.Provider.CmdletProvider.ShouldContinue` method as an additional check for potentially dangerous system modifications.

Attaching Dynamic Parameters to the New-Item Cmdlet

Sometimes the `New-Item` cmdlet requires additional parameters that are specified dynamically at runtime. To provide these dynamic parameters, the container provider must implement the

`System.Management.Automation.Provider.ContainerCmdletProvider.NewItemDynamicParameters*` method. This method retrieves the parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a `System.Management.Automation.RuntimeDefinedParameterDictionary` object. The

Windows PowerShell runtime uses the returned object to add the parameters to the `New-Item` cmdlet.

This provider does not implement this method. However, the following code is the default implementation of this method.

Removing Items

To remove items, the Windows PowerShell provider must override the [System.Management.Automation.Provider.ContainerCmdletProvider.RemoveItem*](#) method to support calls from the `Remove-Item` cmdlet. This method deletes an item from the data store at the specified path. If the `recurse` parameter of the `Remove-Item` cmdlet is set to `true`, the method removes all child items regardless of their level. If the parameter is set to `false`, the method removes only a single item at the specified path.

This provider does not support item removal. However, the following code is the default implementation of

[System.Management.Automation.Provider.ContainerCmdletProvider.RemoveItem*](#).

Things to Remember About Implementing RemoveItem

The following conditions may apply to your implementation of

[System.Management.Automation.Provider.ContainerCmdletProvider.NewItem*](#):

- When defining the provider class, a Windows PowerShell container provider might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of the [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems](#) * method needs to ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude*](#) and [System.Management.Automation.Provider.CmdletProvider.Include*](#) properties.
- By default, overrides of this method should not remove objects unless the [System.Management.Automation.Provider.CmdletProvider.Force*](#) property is set to true. If the specified path indicates a container, the [System.Management.Automation.Provider.CmdletProvider.Force*](#) property is not required.

- Your implementation of `System.Management.Automation.Provider.ContainerCmdletProvider.RemoveItem*` is responsible for preventing infinite recursion when there are circular links, and the like. An appropriate terminating exception should be thrown to reflect such a condition.
- Your implementation of the `System.Management.Automation.Provider.ContainerCmdletProvider.RemoveItem*` method should call `System.Management.Automation.Provider.CmdletProvider.ShouldProcess` and check its return value before making any changes to the data store. After the call to `System.Management.Automation.Provider.CmdletProvider.ShouldProcess` returns `true`, the `System.Management.Automation.Provider.ContainerCmdletProvider.RemoveItem*` method should call the `System.Management.Automation.Provider.CmdletProvider.ShouldContinue` method as an additional check for potentially dangerous system modifications.

Attaching Dynamic Parameters to the Remove-Item Cmdlet

Sometimes the `Remove-Item` cmdlet requires additional parameters that are specified dynamically at runtime. To provide these dynamic parameters, the container provider must implement the

`System.Management.Automation.Provider.ContainerCmdletProvider.RemoveItemDynamicParameters*` method to handle these parameters. This method retrieves the dynamic parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a `System.Management.Automation.RuntimeDefinedParameterDictionary` object. The Windows PowerShell runtime uses the returned object to add the parameters to the `Remove-Item` cmdlet.

This container provider does not implement this method. However, the following code is the default implementation of

`System.Management.Automation.Provider.ContainerCmdletProvider.RemoveItemDynamicParameters*`.

Querying for Child Items

To check to see if child items exist at the specified path, the Windows PowerShell container provider must override the [System.Management.Automation.Provider.ContainerCmdletProvider.HasChildItems*](#) method. This method returns `true` if the item has children, and `false` otherwise. For a null or empty path, the method considers any items in the data store to be children and returns `true`.

Here is the override for the [System.Management.Automation.Provider.ContainerCmdletProvider.HasChildItems*](#) method. If there are more than two path parts created by the `ChunkPath` helper method, the method returns `false`, since only a database container and a table container are defined. For more information about this helper method, see the `ChunkPath` method is discussed in [Creating a Windows PowerShell Item Provider](#).

```
C#  
  
protected override bool HasChildItems( string path )  
{  
    return false;  
} // HasChildItems
```

```
C#  
  
    ErrorCategory.InvalidOperation, tableName));  
}  
  
return results;
```

Things to Remember About Implementing HasChildItems

The following conditions may apply to your implementation of [System.Management.Automation.Provider.ContainerCmdletProvider.HasChildItems*](#):

- If the container provider exposes a root that contains interesting mount points, the implementation of the [System.Management.Automation.Provider.ContainerCmdletProvider.HasChildItems*](#) method should return `true` when a null or an empty string is passed in for the path.

Copying Items

To copy items, the container provider must implement the [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItem](#) method to support calls from the `Copy-Item` cmdlet. This method copies a data item from the location indicated by the `path` parameter of the cmdlet to the location indicated by the `copyPath` parameter. If the `recurse` parameter is specified, the method copies all sub-containers. If the parameter is not specified, the method copies only a single level of items.

This provider does not implement this method. However, the following code is the default implementation of [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItem](#).

Things to Remember About Implementing CopyItem

The following conditions may apply to your implementation of [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItem](#):

- When defining the provider class, a Windows PowerShell container provider might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of the [System.Management.Automation.Provider.ContainerCmdletProvider.GetChildItems](#) * method needs to ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude](#)* and [System.Management.Automation.Provider.CmdletProvider.Include](#)* properties.
- By default, overrides of this method should not copy objects over existing objects unless the [System.Management.Automation.Provider.CmdletProvider.Force](#)* property is set to `true`. For example, the `FileSystem` provider will not copy `C:\temp\abc.txt` over an existing `C:\abc.txt` file unless the [System.Management.Automation.Provider.CmdletProvider.Force](#)* property is set to `true`. If the path specified in the `copyPath` parameter exists and indicates a container, the [System.Management.Automation.Provider.CmdletProvider.Force](#)* property is not required. In this case, [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItem](#) should copy the item indicated by the `path` parameter to the container indicated by the `copyPath` parameter as a child.

- Your implementation of [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItem](#) is responsible for preventing infinite recursion when there are circular links, and the like. An appropriate terminating exception should be thrown to reflect such a condition.
- Your implementation of the [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItem](#) method should call [System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) and check its return value before making any changes to the data store. After the call to [System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) returns true, the [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItem](#) method should call the [System.Management.Automation.Provider.CmdletProvider.ShouldContinue](#) method as an additional check for potentially dangerous system modifications. For more information about calling these methods, see [Rename Items](#).

Attaching Dynamic Parameters to the Copy-Item Cmdlet

Sometimes the `Copy-Item` cmdlet requires additional parameters that are specified dynamically at runtime. To provide these dynamic parameters, the Windows PowerShell container provider must implement the [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItemDynamicParameters*](#) method to handle these parameters. This method retrieves the parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a [System.Management.Automation.RuntimeDefinedParameterDictionary](#) object. The Windows PowerShell runtime uses the returned object to add the parameters to the `Copy-Item` cmdlet.

This provider does not implement this method. However, the following code is the default implementation of [System.Management.Automation.Provider.ContainerCmdletProvider.CopyItemDynamicParameters*](#).

Code Sample

For complete sample code, see [AccessDbProviderSample04 Code Sample](#).

Building the Windows PowerShell Provider

See [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Windows PowerShell Provider

When your Windows PowerShell provider has been registered with Windows PowerShell, you can test it by running the supported cmdlets on the command line. Be aware that the following example output uses a fictitious Access database.

1. Run the `Get-ChildItem` cmdlet to retrieve the list of child items from a Customers table in the Access database.

```
PowerShell  
  
Get-ChildItem mydb:customers
```

The following output appears.

```
Output  
  
PSPPath      : AccessDB::customers  
PSDrive      : mydb  
PSProvider   : System.Management.Automation.ProviderInfo  
PSIsContainer : True  
Data         : System.Data.DataRow  
Name         : Customers  
RowCount     : 91  
Columns      :
```

2. Run the `Get-ChildItem` cmdlet again to retrieve the data of a table.

```
PowerShell  
  
(Get-ChildItem mydb:customers).Data
```

The following output appears.

```
Output  
  
TABLE_CAT    : C:\PS\northwind  
TABLE_SCHEM  :
```

```
TABLE_NAME : Customers
TABLE_TYPE  : TABLE
REMARKS     :
```

3. Now use the `Get-Item` cmdlet to retrieve the items at row 0 in the data table.

```
PowerShell
```

```
Get-Item mydb:\customers\0
```

The following output appears.

```
Output
```

```
PSPath      : AccessDB::customers\0
PSDrive     : mydb
PSProvider   : System.Management.Automation.ProviderInfo
PSIsContainer: False
Data        : System.Data.DataRow
RowNumber    : 0
```

4. Reuse `Get-Item` to retrieve the data for the items in row 0.

```
PowerShell
```

```
(Get-Item mydb:\customers\0).Data
```

The following output appears.

```
Output
```

```
CustomerID   : 1234
CompanyName  : Fabrikam
ContactName   : Eric Gruber
ContactTitle  : President
Address       : 4567 Main Street
City          : Buffalo
Region        : NY
PostalCode    : 98052
Country       : USA
Phone         : (425) 555-0100
Fax           : (425) 555-0101
```

5. Now use the `New-Item` cmdlet to add a row to an existing table. The `Path` parameter specifies the full path to the row, and must indicate a row number that is greater than the existing number of rows in the table. The `Type` parameter

indicates `Row` to specify that type of item to add. Finally, the `Value` parameter specifies a comma-delimited list of column values for the row.

PowerShell

```
New-Item -Path mydb:\Customers\3 -ItemType "Row" -Value  
"3,CustomerFirstName,CustomerLastName,CustomerEmailAddress,CustomerTitle,  
CustomerCompany,CustomerPhone,  
CustomerAddress,CustomerCity,CustomerState,CustomerZip,CustomerCountry"
```

6. Verify the correctness of the new item operation as follows.

none

```
PS mydb:\> cd Customers  
PS mydb:\Customers> (Get-Item 3).Data
```

The following output appears.

Output

```
ID      : 3  
FirstName : Eric  
LastName  : Gruber  
Email    : ericgruber@fabrikam.com  
Title    : President  
Company   : Fabrikam  
WorkPhone : (425) 555-0100  
Address   : 4567 Main Street  
City     : Buffalo  
State    : NY  
Zip      : 98052  
Country   : USA
```

See Also

[Creating Windows PowerShell Providers](#)

[Designing Your Windows PowerShell Provider](#)

[Implementing an Item Windows PowerShell Provider](#)

[Implementing a Navigation Windows PowerShell Provider](#)

[How to Register Cmdlets, Providers, and Host Applications](#)

[Windows PowerShell SDK](#)

Windows PowerShell Programmer's Guide

Creating a Windows PowerShell Navigation Provider

Article • 03/24/2025

This topic describes how to create a Windows PowerShell navigation provider that can navigate the data store. This type of provider supports recursive commands, nested containers, and relative paths.

ⓘ Note

You can download the C# source file (AccessDBSampleProvider05.cs) for this provider using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory. For more information about other Windows PowerShell provider implementations, see [Designing Your Windows PowerShell Provider](#).

The provider described here enables the user handle an Access database as a drive so that the user can navigate to the data tables in the database. When creating your own navigation provider, you can implement methods that can make drive-qualified paths required for navigation, normalize relative paths, move items of the data store, as well as methods that get child names, get the parent path of an item, and test to identify if an item is a container.

ⓘ Caution

Be aware that this design assumes a database that has a field with the name ID, and that the type of the field is LongInteger.

Define the Windows PowerShell provider

A Windows PowerShell navigation provider must create a .NET class that derives from the [System.Management.Automation.Provider.NavigationCmdletProvider](#) base class. Here is the class definition for the navigation provider described in this section.

C#

```
[CmdletProvider("AccessDB", ProviderCapabilities.None)]
public class AccessDBProvider : NavigationCmdletProvider
```

Note that in this provider, the

[System.Management.Automation.Provider.CmdletProviderAttribute](#) attribute includes two parameters. The first parameter specifies a user-friendly name for the provider that is used by Windows PowerShell. The second parameter specifies the Windows PowerShell specific capabilities that the provider exposes to the Windows PowerShell runtime during command processing. For this provider, there are no Windows PowerShell specific capabilities that are added.

Defining Base Functionality

As described in [Design Your PS Provider](#), the [System.Management.Automation.Provider.NavigationCmdletProvider](#) base class derives from several other classes that provided different provider functionality. A Windows PowerShell navigation provider, therefore, must define all of the functionality provided by those classes.

To implement functionality for adding session-specific initialization information and for releasing resources that are used by the provider, see [Creating a Basic PS Provider](#). However, most providers (including the provider described here) can use the default implementation of this functionality provided by Windows PowerShell.

To get access to the data store through a Windows PowerShell drive, you must implement the methods of the [System.Management.Automation.Provider.DriveCmdletProvider](#) base class. For more information about implementing these methods, see [Creating a Windows PowerShell Drive Provider](#).

To manipulate the items of a data store, such as getting, setting, and clearing items, the provider must implement the methods provided by the [System.Management.Automation.Provider.ItemCmdletProvider](#) base class. For more information about implementing these methods, see [Creating a Windows PowerShell Item Provider](#).

To get to the child items, or their names, of the data store, as well as methods that create, copy, rename, and remove items, you must implement the methods provided by the [System.Management.Automation.Provider.ContainerCmdletProvider](#) base class. For more information about implementing these methods, see [Creating a Windows PowerShell Container Provider](#).

Creating a Windows PowerShell Path

Windows PowerShell navigation provider use a provider-internal Windows PowerShell path to navigate the items of the data store. To create a provider-internal path the provider must implement the

[System.Management.Automation.Provider.NavigationCmdletProvider.MakePath*](#)

method to supports calls from the Combine-Path cmdlet. This method combines a parent and child path into a provider-internal path, using a provider-specific path separator between the parent and child paths.

The default implementation takes paths with "/" or "\" as the path separator, normalizes the path separator to "\", combines the parent and child path parts with the separator between them, and then returns a string that contains the combined paths.

This navigation provider does not implement this method. However, the following code is the default implementation of the

[System.Management.Automation.Provider.NavigationCmdletProvider.MakePath*](#) method.

Things to Remember About Implementing MakePath

The following conditions may apply to your implementation of

[System.Management.Automation.Provider.NavigationCmdletProvider.MakePath*:](#)

- Your implementation of the [System.Management.Automation.Provider.NavigationCmdletProvider.MakePath*](#) method should not validate the path as a legal fully-qualified path in the provider namespace. Be aware that each parameter can only represent a part of a path, and the combined parts might not generate a fully-qualified path. For example, the [System.Management.Automation.Provider.NavigationCmdletProvider.MakePath*](#) method for the FileSystem provider might receive "windows\system32" in the `parent` parameter and "abc.dll" in the `child` parameter. The method joins these values with the "\" separator and returns "windows\system32\abc.dll", which is not a fully-qualified file system path.

Important

The path parts provided in the call to

[System.Management.Automation.Provider.NavigationCmdletProvider.MakePath*](#) might contain characters not allowed in the provider namespace. These

characters are most likely used for wildcard expansion and the implementation of this method should not remove them.

Retrieving the Parent Path

Windows PowerShell navigation providers implement the [System.Management.Automation.Provider.NavigationCmdletProvider.GetParentPath*](#) method to retrieve the parent part of the indicated full or partial provider-specific path. The method removes the child part of the path and returns the parent path part. The `root` parameter specifies the fully-qualified path to the root of a drive. This parameter can be null or empty if a mounted drive is not in use for the retrieval operation. If a root is specified, the method must return a path to a container in the same tree as the root.

The sample navigation provider does not override this method, but uses the default implementation. It accepts paths that use both "/" and "\\" as path separators. It first normalizes the path to have only "\\" separators, then splits the parent path off at the last "\\" and returns the parent path.

To Remember About Implementing GetParentPath

Your implementation of the [System.Management.Automation.Provider.NavigationCmdletProvider.GetParentPath*](#) method should split the path lexically on the path separator for the provider namespace. For example, the FileSystem provider uses this method to look for the last "\\" and returns everything to the left of the separator.

Retrieve the Child Path Name

Your navigation provider implements the [System.Management.Automation.Provider.NavigationCmdletProvider.GetChildName*](#) method to retrieve the name (leaf element) of the child of the item located at the indicated full or partial provider-specific path.

The sample navigation provider does not override this method. The default implementation is shown below. It accepts paths that use both "/" and "\\" as path separators. It first normalizes the path to have only "\\" separators, then splits the parent path off at the last "\\" and returns the name of the child path part.

Things to Remember About Implementing GetChildName

Your implementation of the [System.Management.Automation.Provider.NavigationCmdletProvider.GetChildNames*](#) method should split the path lexically on the path separator. If the supplied path contains no path separators, the method should return the path unmodified.

 **Important**

The path provided in the call to this method might contain characters that are illegal in the provider namespace. These characters are most likely used for wildcard expansion or regular expression matching, and the implementation of this method should not remove them.

Determining if an Item is a Container

The navigation provider can implement the [System.Management.Automation.Provider.NavigationCmdletProvider.IsItemContainer*](#) method to determine if the specified path indicates a container. It returns true if the path represents a container, and false otherwise. The user needs this method to be able to use the `Test-Path` cmdlet for the supplied path.

The following code shows the

[System.Management.Automation.Provider.NavigationCmdletProvider.IsItemContainer*](#) implementation in our sample navigation provider. The method verifies that the specified path is correct and if the table exists, and returns true if the path indicates a container.

C#

```
protected override bool IsItemContainer(string path)
{
    if (PathIsDrive(path))
    {
        return true;
    }

    string[] pathChunks = ChunkPath(path);
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out rowNumber);

    if (type == PathType.Table)
    {
        foreach (DatabaseTableInfo ti in GetTables())
        {
            if (ti.Name == tableName)
            {
                return true;
            }
        }
    }
}
```

```
        if (string.Equals(ti.Name, tableName,
 StringComparison.OrdinalIgnoreCase))
    {
        return true;
    }
} // foreach (DatabaseTableInfo...
} // if (pathChunks...

return false;
} // IsItemContainer
```

Things to Remember About Implementing `IsItemContainer`

Your navigation provider .NET class might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In this case, the implementation of [System.Management.Automation.Provider.NavigationCmdletProvider.IsItemContainer*](#) needs to ensure that the path passed meets requirements. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude*](#) property.

Moving an Item

In support of the `Move-Item` cmdlet, your navigation provider implements the [System.Management.Automation.Provider.NavigationCmdletProvider.MoveItem*](#) method. This method moves the item specified by the `path` parameter to the container at the path supplied in the `destination` parameter.

The sample navigation provider does not override the [System.Management.Automation.Provider.NavigationCmdletProvider.MoveItem*](#) method. The following is the default implementation.

Things to Remember About Implementing `MoveItem`

Your navigation provider .NET class might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In this case, the implementation of [System.Management.Automation.Provider.NavigationCmdletProvider.MoveItem*](#) must ensure that the path passed meets requirements. To do this, the method should access the appropriate property, for example, the [CmdletProvider.Exclude](#) property.

By default, overrides of this method should not move objects over existing objects unless the `System.Management.Automation.Provider.CmdletProvider.Force*` property is set to `true`. For example, the `FileSystem` provider will not copy `C:\temp\abc.txt` over an existing `C:\bar.txt` file unless the

`System.Management.Automation.Provider.CmdletProvider.Force*` property is set to `true`. If the path specified in the `destination` parameter exists and is a container, the `System.Management.Automation.Provider.CmdletProvider.Force*` property is not required. In this case,

`System.Management.Automation.Provider.NavigationCmdletProvider.MoveItem*` should move the item indicated by the `path` parameter to the container indicated by the `destination` parameter as a child.

Your implementation of the

`System.Management.Automation.Provider.NavigationCmdletProvider.MoveItem*` method should call

`System.Management.Automation.Provider.CmdletProvider.ShouldProcess` and check its return value before making any changes to the data store. This method is used to confirm execution of an operation when a change is made to system state, for example, deleting files. `System.Management.Automation.Provider.CmdletProvider.ShouldProcess` sends the name of the resource to be changed to the user, with the Windows PowerShell runtime taking into account any command line settings or preference variables in determining what should be displayed to the user.

After the call to

`System.Management.Automation.Provider.CmdletProvider.ShouldProcess` returns `true`, the `System.Management.Automation.Provider.NavigationCmdletProvider.MoveItem*` method should call the

`System.Management.Automation.Provider.CmdletProvider.ShouldContinue` method. This method sends a message to the user to allow feedback to say if the operation should be continued. Your provider should call

`System.Management.Automation.Provider.CmdletProvider.ShouldContinue` as an additional check for potentially dangerous system modifications.

Attaching Dynamic Parameters to the Move-Item Cmdlet

Sometimes the `Move-Item` cmdlet requires additional parameters that are provided dynamically at runtime. To provide these dynamic parameters, the navigation provider must implement the

`System.Management.Automation.Provider.NavigationCmdletProvider.MoveItemDynamic`

[Parameters*](#) method to get the required parameter values from the item at the indicated path, and return an object that has properties and fields with parsing attributes similar to a cmdlet class or a [System.Management.Automation.RuntimeDefinedParameterDictionary](#) object.

This navigation provider does not implement this method. However, the following code is the default implementation of [System.Management.Automation.Provider.NavigationCmdletProvider.MoveItemDynamicParameters*](#).

Normalizing a Relative Path

Your navigation provider implements the [System.Management.Automation.Provider.NavigationCmdletProvider.NormalizeRelativePath*](#) method to normalize the fully-qualified path indicated in the `path` parameter as being relative to the path specified by the `basePath` parameter. The method returns a string representation of the normalized path. It writes an error if the `path` parameter specifies a nonexistent path.

The sample navigation provider does not override this method. The following is the default implementation.

Things to Remember About Implementing NormalizeRelativePath

Your implementation of [System.Management.Automation.Provider.NavigationCmdletProvider.NormalizeRelativePath*](#) should parse the `path` parameter, but it does not have to use purely syntactical parsing. You are encouraged to design this method to use the path to look up the path information in the data store and create a path that matches the casing and standardized path syntax.

Code Sample

For complete sample code, see [AccessDbProviderSample05 Code Sample](#).

Defining Object Types and Formatting

It is possible for a provider to add members to existing objects or define new objects. For more information, see [Extending Object Types and Formatting](#).

Building the Windows PowerShell provider

For more information, see [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Windows PowerShell provider

When your Windows PowerShell provider has been registered with Windows PowerShell, you can test it by running the supported cmdlets on the command line, including cmdlets made available by derivation. This example will test the sample navigation provider.

1. Run your new shell and use the `Set-Location` cmdlet to set the path to indicate the Access database.

PowerShell

```
Set-Location mydb:
```

2. Now run the `Get-ChildItem` cmdlet to retrieve a list of the database items, which are the available database tables. For each table, this cmdlet also retrieves the number of table rows.

PowerShell

```
Get-ChildItem | Format-Table RowCount, Name -AutoSize
```

Output

RowCount	Name
-----	-----
180	MSysAccessObjects
0	MSysACEs
1	MSysCmdbars
0	MSysIMEXColumns
0	MSysIMEXSpecs
0	MSysObjects
0	MSysQueries
7	MSysRelationships
8	Categories
91	Customers
9	Employees
2155	Order Details
830	Orders
77	Products

```
3    Shippers  
29   Suppliers
```

3. Use the `Set-Location` cmdlet again to set the location of the Employees data table.

```
PowerShell  
  
Set-Location Employees
```

4. Let's now use the `Get-Location` cmdlet to retrieve the path to the Employees table.

```
PowerShell  
  
Get-Location  
  
Output  
  
Path  
----  
mydb:\Employees
```

5. Now use the `Get-ChildItem` cmdlet piped to the `Format-Table` cmdlet. This set of cmdlets retrieves the items for the Employees data table, which are the table rows. They are formatted as specified by the `Format-Table` cmdlet.

```
PowerShell  
  
Get-ChildItem | Format-Table RowNumber, PSIsContainer, Data -AutoSize
```

```
Output  
  
RowNumber PSIsContainer Data  
----- -----  
0          False        System.Data.DataRow  
1          False        System.Data.DataRow  
2          False        System.Data.DataRow  
3          False        System.Data.DataRow  
4          False        System.Data.DataRow  
5          False        System.Data.DataRow  
6          False        System.Data.DataRow  
7          False        System.Data.DataRow  
8          False        System.Data.DataRow
```

6. You can now run the `Get-Item` cmdlet to retrieve the items for row 0 of the Employees data table.

PowerShell

```
Get-Item 0
```

Output

```
PSPath      : AccessDB::C:\PS\Northwind.mdb\Employees\0
PSParentPath : AccessDB::C:\PS\Northwind.mdb\Employees
PSChildName  : 0
PSDrive      : mydb
PSProvider    : System.Management.Automation.ProviderInfo
PSIsContainer : False
Data         : System.Data.DataRow
RowNumber     : 0
```

7. Use the `Get-Item` cmdlet again to retrieve the employee data for the items in row 0.

PowerShell

```
(Get-Item 0).Data
```

Output

```
EmployeeID   : 1
LastName     : Davis
FirstName    : Sara
Title        : Sales Representative
TitleOfCourtesy : Ms.
BirthDate    : 12/8/1968 12:00:00 AM
HireDate     : 5/1/1992 12:00:00 AM
Address       : 4567 Main Street
                  Apt. 2A
City          : Buffalo
Region        : NY
PostalCode    : 98052
Country       : USA
HomePhone     : (206) 555-9857
Extension     : 5467
Photo          : EmpID1.bmp
Notes          : Education includes a BA in psychology from
                  Colorado State University. She also completed "The
                  Art of the Cold Call." Nancy is a member of
                  Toastmasters International.
ReportsTo     : 2
```

See Also

[Creating Windows PowerShell providers](#)

[Design Your Windows PowerShell provider](#)

[Extending Object Types and Formatting](#)

[Implement a Container Windows PowerShell provider](#)

[How to Register Cmdlets, Providers, and Host Applications](#)

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

Creating a Windows PowerShell Content Provider

Article • 03/24/2025

This topic describes how to create a Windows PowerShell provider that enables the user to manipulate the contents of the items in a data store. As a consequence, a provider that can manipulate the contents of items is referred to as a Windows PowerShell content provider.

ⓘ Note

You can download the C# source file (AccessDBSampleProvider06.cs) for this provider using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory. For more information about other Windows PowerShell provider implementations, see [Designing Your Windows PowerShell Provider](#).

Define the Windows PowerShell Content Provider Class

A Windows PowerShell content provider must create a .NET class that supports the [System.Management.Automation.Provider.IContentCmdletProvider](#) interface. Here is the class definition for the item provider described in this section.

C#

```
[CmdletProvider("AccessDB", ProviderCapabilities.None)]
public class AccessDBProvider : NavigationCmdletProvider,
IContentCmdletProvider
```

Note that in this class definition, the [System.Management.Automation.Provider.CmdletProviderAttribute](#) attribute includes two parameters. The first parameter specifies a user-friendly name for the provider that is used by Windows PowerShell. The second parameter specifies the Windows PowerShell specific capabilities that the provider exposes to the Windows PowerShell

runtime during command processing. For this provider, there are no added Windows PowerShell specific capabilities.

Define Functionality of Base Class

As described in [Design Your Windows PowerShell Provider](#), the `System.Management.Automation.Provider.NavigationCmdletProvider` class derives from several other classes that provided different provider functionality. A Windows PowerShell content provider, therefore, typically defines all of the functionality provided by those classes.

For more information about how to implement functionality for adding session-specific initialization information and for releasing resources that are used by the provider, see [Creating a Basic Windows PowerShell Provider](#). However, most providers, including the provider described here, can use the default implementation of this functionality that is provided by Windows PowerShell.

To access the data store, the provider must implement the methods of the `System.Management.Automation.Provider.DriveCmdletProvider` base class. For more information about implementing these methods, see [Creating a Windows PowerShell Drive Provider](#).

To manipulate the items of a data store, such as getting, setting, and clearing items, the provider must implement the methods provided by the `System.Management.Automation.Provider.ItemCmdletProvider` base class. For more information about implementing these methods, see [Creating a Windows PowerShell Item Provider](#).

To work on multi-layer data stores, the provider must implement the methods provided by the `System.Management.Automation.Provider.ContainerCmdletProvider` base class. For more information about implementing these methods, see [Creating a Windows PowerShell Container Provider](#).

To support recursive commands, nested containers, and relative paths, the provider must implement the `System.Management.Automation.Provider.NavigationCmdletProvider` base class. In addition, this Windows PowerShell content provider can attach the `System.Management.Automation.Provider.IContentCmdletProvider` interface to the `System.Management.Automation.Provider.NavigationCmdletProvider` base class, and must therefore implement the methods provided by that class. For more information, see implementing those methods, see [Implement a Navigation Windows PowerShell Provider](#).

Implementing a Content Reader

To read content from an item, a provider must implements a content reader class that derives from [System.Management.Automation.Provider.IContentReader](#). The content reader for this provider allows access to the contents of a row in a data table. The content reader class defines a **Read** method that retrieves the data from the indicated row and returns a list representing that data, a **Seek** method that moves the content reader, a **Close** method that closes the content reader, and a **Dispose** method.

C#

```
public class AccessDBContentReader : IContentReader
{
    // A provider instance is required so as to get "content"
    private AccessDBProvider provider;
    private string path;
    private long currentOffset;

    internal AccessDBContentReader(string path, AccessDBProvider provider)
    {
        this.path = path;
        this.provider = provider;
    }

    /// <summary>
    /// Read the specified number of rows from the source.
    /// </summary>
    /// <param name="readCount">The number of items to
    /// return.</param>
    /// <returns>An array of elements read.</returns>
    public IList Read(long readCount)
    {
        // Read the number of rows specified by readCount and increment
        // offset
        string tableName;
        int lineNumber;
        PathType type = provider.GetNamesFromPath(path, out tableName, out
rowNumber);

        Collection<DatabaseRowInfo> rows =
            provider.GetRows(tableName);
        Collection<DataRow> results = new Collection<DataRow>();

        if (currentOffset < 0 || currentOffset >= rows.Count)
        {
            return null;
        }

        int rowsRead = 0;

        while (rowsRead < readCount && currentOffset < rows.Count)
```

```

        {
            results.Add(rows[(int)currentOffset].Data);
            rowsRead++;
            currentOffset++;
        }

        return results;
    } // Read

    /// <summary>
    /// Moves the content reader specified number of rows from the
    /// origin
    /// </summary>
    /// <param name="offset">Number of rows to offset</param>
    /// <param name="origin">Starting row from which to offset</param>
    public void Seek(long offset, System.IO.SeekOrigin origin)
    {
        // get the number of rows in the table which will help in
        // calculating current position
        string tableName;
        int rowCount;

        PathType type = provider.GetNamesFromPath(path, out tableName, out
rowCount);

        if (type == PathType.Invalid)
        {
            throw new ArgumentException("Path specified must represent a
table or a row :" + path);
        }

        if (type == PathType.Table)
        {
            Collection<DatabaseRowInfo> rows = provider.GetRows(tableName);

            int numRows = rows.Count;

            if (offset > rows.Count)
            {
                throw new
                    ArgumentException(
                        "Offset cannot be greater than the number of rows
available"
                    );
            }

            if (origin == System.IO.SeekOrigin.Begin)
            {
                // starting from Beginning with an index 0, the current
                offset
                // has to be advanced to offset - 1
                currentOffset = offset - 1;
            }
            else if (origin == System.IO.SeekOrigin.End)
            {

```

```

        // starting from the end which is numRows - 1, the current
        // offset is so much less than numRows - 1
        currentOffset = numRows - 1 - offset;
    }
    else
    {
        // calculate from the previous value of current offset
        // advancing forward always
        currentOffset += offset;
    }
} // if (type...
else
{
    // for row, the offset will always be set to 0
    currentOffset = 0;
}

} // Seek

/// <summary>
/// Closes the content reader, so all members are reset
/// </summary>
public void Close()
{
    Dispose();
} // Close

/// <summary>
/// Dispose any resources being used
/// </summary>
public void Dispose()
{
    Seek(0, System.IO.SeekOrigin.Begin);

    GC.SuppressFinalize(this);
} // Dispose
} // AccessDBContentReader

```

Implementing a Content Writer

To write content to an item, a provider must implement a content writer class derives from [System.Management.Automation.Provider.IContentWriter](#). The content writer class defines a **Write** method that writes the specified row content, a **Seek** method that moves the content writer, a **Close** method that closes the content writer, and a **Dispose** method.

C#

```

public class AccessDBContentWriter : IContentWriter
{

```

```
// A provider instance is required so as to get "content"
private AccessDBProvider provider;
private string path;
private long currentOffset;

internal AccessDBContentWriter(string path, AccessDBProvider provider)
{
    this.path = path;
    this.provider = provider;
}

/// <summary>
/// Write the specified row contents in the source
/// </summary>
/// <param name="content"> The contents to be written to the source.
/// </param>
/// <returns>An array of elements which were successfully written to
/// the source</returns>
///
public IList Write(IList content)
{
    if (content == null)
    {
        return null;
    }

    // Get the total number of rows currently available it will
    // determine how much to overwrite and how much to append at
    // the end
    string tableName;
    int rowNumber;
    PathType type = provider.GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        OdbcDataAdapter da = provider.GetAdapterForTable(tableName);
        if (da == null)
        {
            return null;
        }

        DataSet ds = provider.GetDataSetForTable(da, tableName);
        DataTable table = provider.GetDataTable(ds, tableName);

        string[] colValues = (content[0] as string).Split(',');
        // set the specified row
        DataRow row = table.NewRow();

        for (int i = 0; i < colValues.Length; i++)
        {
            if (!String.IsNullOrEmpty(colValues[i]))
            {
                row[i] = colValues[i];
            }
        }

        table.Rows.Add(row);
        provider.UpdateTable(table, tableName);
    }
}
```

```

        }

        //table.Rows.InsertAt(row, rowNumber);
        // Update the table
        table.Rows.Add(row);
        da.Update(ds, tableName);

    }
    else
    {
        throw new InvalidOperationException("Operation not supported.
Content can be added only for tables");
    }

    return null;
} // Write

/// <summary>
/// Moves the content reader specified number of rows from the
/// origin
/// </summary>
/// <param name="offset">Number of rows to offset</param>
/// <param name="origin">Starting row from which to offset</param>
public void Seek(long offset, System.IO.SeekOrigin origin)
{
    // get the number of rows in the table which will help in
    // calculating current position
    string tableName;
    int rowNumber;

    PathType type = provider.GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Invalid)
    {
        throw new ArgumentException("Path specified should represent
either a table or a row : " + path);
    }

    Collection<DatabaseRowInfo> rows =
        provider.GetRows(tableName);

    int numRows = rows.Count;

    if (offset > rows.Count)
    {
        throw new
            ArgumentException(
                "Offset cannot be greater than the number of rows
available"
            );
    }

    if (origin == System.IO.SeekOrigin.Begin)

```

```

    {
        // starting from Beginning with an index 0, the current offset
        // has to be advanced to offset - 1
        currentOffset = offset - 1;
    }
    else if (origin == System.IO.SeekOrigin.End)
    {
        // starting from the end which is numRows - 1, the current
        // offset is so much less than numRows - 1
        currentOffset = numRows - 1 - offset;
    }
    else
    {
        // calculate from the previous value of current offset
        // advancing forward always
        currentOffset += offset;
    }
}

} // Seek

/// <summary>
/// Closes the content reader, so all members are reset
/// </summary>
public void Close()
{
    Dispose();
} // Close

/// <summary>
/// Dispose any resources being used
/// </summary>
public void Dispose()
{
    Seek(0, System.IO.SeekOrigin.Begin);

    GC.SuppressFinalize(this);
} // Dispose
} // AccessDBContentWriter

```

Retrieving the Content Reader

To get content from an item, the provider must implement the [System.Management.Automation.Provider.IContentCmdletProvider.GetContentReader*](#) to support the `Get-Content` cmdlet. This method returns the content reader for the item located at the specified path. The reader object can then be opened to read the content.

Here is the implementation of

[System.Management.Automation.Provider.IContentCmdletProvider.GetContentReader*](#) for this method for this provider.

C#

```
public IContentReader GetContentReader(string path)
{
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out rowNumber);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }
    else if (type == PathType.Row)
    {
        throw new InvalidOperationException("contents can be obtained only
for tables");
    }

    return new AccessDBContentReader(path, this);
} // GetContentReader
```

C#

```
public IContentReader GetContentReader(string path)
{
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out rowNumber);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }
    else if (type == PathType.Row)
    {
        throw new InvalidOperationException("contents can be obtained only
for tables");
    }

    return new AccessDBContentReader(path, this);
} // GetContentReader
```

Things to Remember About Implementing GetContentReader

The following conditions may apply to an implementation of
`System.Management.Automation.Provider.IContentCmdletProvider.GetContentReader`*:

- When defining the provider class, a Windows PowerShell content provider might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the `System.Management.Automation.Provider.ProviderCapabilities` enumeration. In these cases, the implementation of the `System.Management.Automation.Provider.IContentCmdletProvider.GetContentReader*` method must ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the `System.Management.Automation.Provider.CmdletProvider.Exclude*` and `System.Management.Automation.Provider.CmdletProvider.Include*` properties.
- By default, overrides of this method should not retrieve a reader for objects that are hidden from the user unless the `System.Management.Automation.Provider.CmdletProvider.Force*` property is set to `true`. An error should be written if the path represents an item that is hidden from the user and `System.Management.Automation.Provider.CmdletProvider.Force*` is set to `false`.

Attaching Dynamic Parameters to the Get-Content Cmdlet

The `Get-Content` cmdlet might require additional parameters that are specified dynamically at runtime. To provide these dynamic parameters, the Windows PowerShell content provider must implement the `System.Management.Automation.Provider.IContentCmdletProvider.GetContentReaderDynamicParameters*` method. This method retrieves dynamic parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a `System.Management.Automation.RuntimeDefinedParameterDictionary` object. The Windows PowerShell runtime uses the returned object to add the parameters to the cmdlet.

This Windows PowerShell container provider does not implement this method. However, the following code is the default implementation of this method.

C#

```
public object GetContentReaderDynamicParameters(string path)
{
    return null;
}
```

```
C#
```

```
public object GetContentReaderDynamicParameters(string path)
{
    return null;
}
```

Retrieving the Content Writer

To write content to an item, the provider must implement the [System.Management.Automation.Provider.IContentCmdletProvider.GetContentWriter*](#) to support the `Set-Content` and `Add-Content` cmdlets. This method returns the content writer for the item located at the specified path.

Here is the implementation of [System.Management.Automation.Provider.IContentCmdletProvider.GetContentWriter*](#) for this method.

```
C#
```

```
public IContentWriter GetContentWriter(string path)
{
    string tableName;
    int rowCount;

    PathType type = GetNamesFromPath(path, out tableName, out rowCount);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidOperationException(path);
    }
    else if (type == PathType.Row)
    {
        throw new InvalidOperationException("contents can be added only to
tables");
    }

    return new AccessDBContentWriter(path, this);
}
```

```
C#
```

```
public IContentWriter GetContentWriter(string path)
{
    string tableName;
    int rowCount;

    PathType type = GetNamesFromPath(path, out tableName, out rowCount);
```

```

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidOperationException(path);
    }
    else if (type == PathType.Row)
    {
        throw new InvalidOperationException("contents can be added only to
tables");
    }

    return new AccessDBContentWriter(path, this);
}

```

Things to Remember About Implementing GetContentWriter

The following conditions may apply to your implementation of [System.Management.Automation.Provider.IContentCmdletProvider.GetContentWriter*](#):

- When defining the provider class, a Windows PowerShell content provider might declare provider capabilities of ExpandWildcards, Filter, Include, or Exclude, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of the [System.Management.Automation.Provider.IContentCmdletProvider.GetContentWriter*](#) method must ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude*](#) and [System.Management.Automation.Provider.CmdletProvider.Include*](#) properties.
- By default, overrides of this method should not retrieve a writer for objects that are hidden from the user unless the [System.Management.Automation.Provider.CmdletProvider.Force*](#) property is set to `true`. An error should be written if the path represents an item that is hidden from the user and [System.Management.Automation.Provider.CmdletProvider.Force*](#) is set to `false`.

Attaching Dynamic Parameters to the Add-Content and Set-Content Cmdlets

The `Add-Content` and `Set-Content` cmdlets might require additional dynamic parameters that are added at runtime. To provide these dynamic parameters, the Windows PowerShell content provider must implement the

`System.Management.Automation.Provider.IContentCmdletProvider.GetContentWriterDynamicParameters*` method to handle these parameters. This method retrieves dynamic parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a `System.Management.Automation.RuntimeDefinedParameterDictionary` object. The Windows PowerShell runtime uses the returned object to add the parameters to the cmdlets.

This Windows PowerShell container provider does not implement this method. However, the following code is the default implementation of this method.

```
C#
```

```
public object GetContentWriterDynamicParameters(string path)
{
    return null;
}
```

Clearing Content

Your content provider implements the `System.Management.Automation.Provider.IContentCmdletProvider.ClearContent*` method in support of the `clear-Content` cmdlet. This method removes the contents of the item at the specified path, but leaves the item intact.

Here is the implementation of the `System.Management.Automation.Provider.IContentCmdletProvider.ClearContent*` method for this provider.

```
C#
```

```
public void ClearContent(string path)
{
    string tableName;
    int rowCount;

    PathType type = GetNamesFromPath(path, out tableName, out rowCount);

    if (type != PathType.Table)
    {
        WriteError(new ErrorRecord(
            new InvalidOperationException("Operation not supported. Content
can be cleared only for table"),
            "NotValidRow", ErrorCategory.InvalidArgument,
            path));
    }
}
```

```

}

OdbcDataAdapter da = GetAdapterForTable(tableName);

if (da == null)
{
    return;
}

DataSet ds = GetDataSetForTable(da, tableName);
DataTable table = GetDataTable(ds, tableName);

// Clear contents at the specified location
for (int i = 0; i < table.Rows.Count; i++)
{
    table.Rows[i].Delete();
}

if (ShouldProcess(path, "ClearContent"))
{
    da.Update(ds, tableName);
}

} // ClearContent

```

Things to Remember About Implementing `ClearContent`

The following conditions may apply to an implementation of [System.Management.Automation.Provider.IContentCmdletProvider.ClearContent*](#):

- When defining the provider class, a Windows PowerShell content provider might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of the [System.Management.Automation.Provider.IContentCmdletProvider.ClearContent*](#) method must ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude*](#) and [System.Management.Automation.Provider.CmdletProvider.Include*](#) properties.
- By default, overrides of this method should not clear the contents of objects that are hidden from the user unless the [System.Management.Automation.Provider.CmdletProvider.Force*](#) property is set to `true`. An error should be written if the path represents an item that is hidden from the user and [System.Management.Automation.Provider.CmdletProvider.Force*](#) is set to `false`.

- Your implementation of the [System.Management.Automation.Provider.IContentCmdletProvider.ClearContent*](#) method should call [System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) and verify its return value before making any changes to the data store. This method is used to confirm execution of an operation when a change is made to the data store, such as clearing content. The [System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) method sends the name of the resource to be changed to the user, with the Windows PowerShell runtime handling any command-line settings or preference variables in determining what should be displayed.

After the call to

[System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) returns `true`, the

[System.Management.Automation.Provider.IContentCmdletProvider.ClearContent*](#) method should call the

[System.Management.Automation.Provider.CmdletProvider.ShouldContinue](#) method. This method sends a message to the user to allow feedback to verify if the operation should be continued. The call to

[System.Management.Automation.Provider.CmdletProvider.ShouldContinue](#) allows an additional check for potentially dangerous system modifications.

Attaching Dynamic Parameters to the Clear-Content Cmdlet

The `Clear-Content` cmdlet might require additional dynamic parameters that are added at runtime. To provide these dynamic parameters, the Windows PowerShell content provider must implement the

[System.Management.Automation.Provider.IContentCmdletProvider.ClearContentDynamicParameters*](#) method to handle these parameters. This method retrieves the parameters for the item at the indicated path. This method retrieves dynamic parameters for the item at the indicated path and returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a

[System.Management.Automation.RuntimeDefinedParameterDictionary](#) object. The Windows PowerShell runtime uses the returned object to add the parameters to the cmdlet.

This Windows PowerShell container provider does not implement this method. However, the following code is the default implementation of this method.

```
C#
```

```
public object ClearContentDynamicParameters(string path)
{
    return null;
}
```

```
C#
```

```
public object ClearContentDynamicParameters(string path)
{
    return null;
}
```

Code Sample

For complete sample code, see [AccessDbProviderSample06 Code Sample](#).

Defining Object Types and Formatting

When writing a provider, it may be necessary to add members to existing objects or define new objects. When this is done, you must create a Types file that Windows PowerShell can use to identify the members of the object and a Format file that defines how the object is displayed. For more information, see [Extending Object Types and Formatting](#).

Building the Windows PowerShell Provider

See [How to Register Cmdlets, Providers, and Host Applications](#).

Testing the Windows PowerShell Provider

When your Windows PowerShell provider has been registered with Windows PowerShell, you can test it by running the supported cmdlets on the command line. For example, test the sample content provider.

Use the `Get-Content` cmdlet to retrieve the contents of specified item in the database table at the path specified by the `Path` parameter. The `ReadCount` parameter specifies the number of items for the defined content reader to read (default 1). With the following command entry, the cmdlet retrieves two rows (items) from the table and

displays their contents. Note that the following example output uses a fictitious Access database.

PowerShell

```
Get-Content -Path mydb:\Customers -ReadCount 2
```

Output

```
ID      : 1
FirstName : Eric
LastName  : Gruber
Email    : ericgruber@fabrikam.com
Title    : President
Company  : Fabrikam
WorkPhone : (425) 555-0100
Address   : 4567 Main Street
City     : Buffalo
State    : NY
Zip      : 98052
Country   : USA
ID      : 2
FirstName : Eva
LastName  : Corets
Email    : evacorets@cohowinery.com
Title    : Sales Representative
Company  : Coho Winery
WorkPhone : (360) 555-0100
Address   : 8910 Main Street
City     : Cabmerlot
State    : WA
Zip      : 98089
Country   : USA
```

See Also

[Creating Windows PowerShell providers](#)

[Design Your Windows PowerShell provider](#)

[Extending Object Types and Formatting ↗](#)

[Implement a Navigation Windows PowerShell provider](#)

[How to Register Cmdlets, Providers, and Host Applications](#)

[Windows PowerShell SDK](#)

Windows PowerShell Programmer's Guide

Creating a Windows PowerShell Property Provider

Article • 03/24/2025

This topic describes how to create a provider that enables the user to manipulate the properties of items in a data store. As a consequence, this type of provider is referred to as a Windows PowerShell property provider. For example, the Registry provider provided by Windows PowerShell handles registry key values as properties of the registry key item. This type of provider must add the

`System.Management.Automation.Provider.IPropertyCmdletProvider` interface to the implementation of the .NET class.

ⓘ Note

Windows PowerShell provides a template file that you can use to develop a Windows PowerShell provider. The `TemplateProvider.cs` file is available on the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded template is available in the <PowerShell Samples> directory. You should make a copy of this file and use the copy for creating a new Windows PowerShell provider, removing any functionality that you do not need. For more information about other Windows PowerShell provider implementations, see [Designing Your Windows PowerShell Provider](#).

✖ Caution

The methods of your property provider should write any objects using the [`System.Management.Automation.Provider.CmdletProvider.Writepropertyobject*`](#) method.

Defining the Windows PowerShell provider

A property provider must create a .NET class that supports the `System.Management.Automation.Provider.IPropertyCmdletProvider` interface. Here is the default class declaration from the `TemplateProvider.cs` file provided by Windows PowerShell.

Defining Base Functionality

The [System.Management.Automation.Provider.IPropertyCmdletProvider](#) interface can be attached to any of the provider base classes, with the exception of the [System.Management.Automation.Provider.DriveCmdletProvider](#) class. Add the base functionality that is required by the base class you are using. For more information about base classes, see [Designing Your Windows PowerShell Provider](#).

Retrieving Properties

To retrieve properties, the provider must implement the [System.Management.Automation.Provider.IPropertyCmdletProvider.GetProperty*](#) method to support calls from the `Get-ItemProperty` cmdlet. This method retrieves the properties of the item located at the specified provider-internal path (fully-qualified).

The `providerSpecificPickList` parameter indicates which properties to retrieve. If this parameter is `null` or empty, the method should retrieve all properties. In addition, [System.Management.Automation.Provider.IPropertyCmdletProvider.GetProperty*](#) writes an instance of a [System.Management.Automation.PSObject](#) object that represents a property bag of the retrieved properties. The method should return nothing.

It is recommended that the implementation of [System.Management.Automation.Provider.IPropertyCmdletProvider.GetProperty*](#) supports the wildcard expansion of property names for each element in the pick list. To do this, use the [System.Management.Automation.WildcardPattern](#) class to perform the wildcard pattern matching.

Here is the default implementation of [System.Management.Automation.Provider.IPropertyCmdletProvider.GetProperty*](#) from the `TemplateProvider.cs` file provided by Windows PowerShell.

Things to Remember About Implementing GetProperty

The following conditions may apply to your implementation of [System.Management.Automation.Provider.IPropertyCmdletProvider.GetProperty*](#):

- When defining the provider class, a Windows PowerShell property provider might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of the [System.Management.Automation.Provider.IPropertyCmdletProvider.GetProperty*](#)

method needs to ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude*](#) and [System.Management.Automation.Provider.CmdletProvider.Include*](#) properties.

- By default, overrides of this method should not retrieve a reader for objects that are hidden from the user unless the [System.Management.Automation.Provider.CmdletProvider.Force*](#) property is set to `true`. An error should be written if the path represents an item that is hidden from the user and [System.Management.Provider.CmdletProvider.Force*](#) is set to `false`.

Attaching Dynamic Parameters to the Get-ItemProperty Cmdlet

The `Get-ItemProperty` cmdlet might require additional parameters that are specified dynamically at runtime. To provide these dynamic parameters, the Windows PowerShell property provider must implement the [System.Management.Automation.Provider.IPropertyCmdletProvider.GetPropertyDynamicParameters*](#) method. The `path` parameter indicates a fully-qualified provider-internal path, while the `providerSpecificPickList` parameter specifies the provider-specific properties entered on the command line. This parameter might be `null` or empty if the properties are piped to the cmdlet. In this case, this method returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a [System.Management.Automation.RuntimeDefinedParameterDictionary](#) object. The Windows PowerShell runtime uses the returned object to add the parameters to the cmdlet.

Here is the default implementation of [System.Management.Automation.Provider.IPropertyCmdletProvider.GetPropertyDynamicParameters*](#) from the `TemplateProvider.cs` file provided by Windows PowerShell.

Setting Properties

To set properties, the Windows PowerShell property provider must implement the [System.Management.Automation.Provider.IPropertyCmdletProvider SetProperty*](#) method to support calls from the `Set-ItemProperty` cmdlet. This method sets one or more properties of the item at the specified path, and overwrites the supplied properties as required.

[System.Management.Automation.Provider.IPropertyCmdletProvider SetProperty](#)* also writes an instance of a [System.Management.Automation.PSObject](#) object that represents a property bag of the updated properties.

Here is the default implementation of

[System.Management.Automation.Provider.IPropertyCmdletProvider SetProperty](#)* from the TemplateProvider.cs file provided by Windows PowerShell.

Things to Remember About Implementing Set-ItemProperty

The following conditions may apply to an implementation of

[System.Management.Automation.Provider.IPropertyCmdletProvider SetProperty](#)*:

- When defining the provider class, a Windows PowerShell property provider might declare provider capabilities of ExpandWildcards, Filter, Include, or Exclude, from the [System.Management.Automation.Provider.ProviderCapabilities](#) enumeration. In these cases, the implementation of the [System.Management.Automation.Provider.IPropertyCmdletProvider SetProperty](#)* method must ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the [System.Management.Automation.Provider.CmdletProvider.Exclude](#)* and [System.Management.Automation.Provider.CmdletProvider.Include](#)* properties.
- By default, overrides of this method should not retrieve a reader for objects that are hidden from the user unless the [System.Management.Automation.Provider.CmdletProvider.Force](#)* property is set to `true`. An error should be written if the path represents an item that is hidden from the user and [System.Management.Automation.Provider.CmdletProvider.Force](#)* is set to `false`.
- Your implementation of the [System.Management.Automation.Provider.IPropertyCmdletProvider SetProperty](#)* method should call [System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) and verify its return value before making any changes to the data store. This method is used to confirm execution of an operation when a change is made to system state, for example, renaming files. [System.Management.Automation.Provider.CmdletProvider.ShouldProcess](#) sends the name of the resource to be changed to the user, with the Windows PowerShell runtime and handling any command-line settings or preference variables in determining what should be displayed.

After the call to `System.Management.Automation.Provider.CmdletProvider.ShouldProcess` returns `true`, if potentially dangerous system modifications can be made, the `System.Management.Automation.Provider.IPropertyCmdletProvider SetProperty*` method should call the `System.Management.Automation.Provider.CmdletProvider.ShouldContinue` method. This method sends a confirmation message to the user to allow additional feedback to indicate that the operation should be continued.

Attaching Dynamic Parameters for the Set-ItemProperty Cmdlet

The `Set-ItemProperty` cmdlet might require additional parameters that are specified dynamically at runtime. To provide these dynamic parameters, the Windows PowerShell property provider must implement the `System.Management.Automation.Provider.IPropertyCmdletProvider SetPropertyDynamicParameters*` method. This method returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a `System.Management.Automation.RuntimeDefinedParameterDictionary` object. The `null` value can be returned if no dynamic parameters are to be added.

Here is the default implementation of `System.Management.Automation.Provider.IPropertyCmdletProvider GetPropertyDynamicParameters*` from the `TemplateProvider.cs` file provided by Windows PowerShell.

Clearing Properties

To clear properties, the Windows PowerShell property provider must implement the `System.Management.Automation.Provider.IPropertyCmdletProvider.ClearProperty*` method to support calls from the `Clear-ItemProperty` cmdlet. This method sets one or more properties for the item located at the specified path.

Here is the default implementation of `System.Management.Automation.Provider.IPropertyCmdletProvider.ClearProperty*` from the `TemplateProvider.cs` file provided by Windows PowerShell.

Thing to Remember About Implementing ClearProperty

The following conditions may apply to your implementation of `System.Management.Automation.Provider.IPropertyCmdletProvider.ClearProperty*`:

- When defining the provider class, a Windows PowerShell property provider might declare provider capabilities of `ExpandWildcards`, `Filter`, `Include`, or `Exclude`, from the `System.Management.Automation.Provider.ProviderCapabilities` enumeration. In these cases, the implementation of the `System.Management.Automation.Provider.IPropertyCmdletProvider.ClearProperty*` method needs to ensure that the path passed to the method meets the requirements of the specified capabilities. To do this, the method should access the appropriate property, for example, the `System.Management.Automation.Provider.CmdletProvider.Exclude*` and `System.Management.Automation.Provider.CmdletProvider.Include*` properties.
- By default, overrides of this method should not retrieve a reader for objects that are hidden from the user unless the `System.Management.Automation.Provider.CmdletProvider.Force*` property is set to `true`. An error should be written if the path represents an item that is hidden from the user and `System.Management.Automation.Provider.CmdletProvider.Force*` is set to `false`.
- Your implementation of the `System.Management.Automation.Provider.IPropertyCmdletProvider.ClearProperty*` method should call `System.Management.Automation.Provider.CmdletProvider.ShouldProcess` and verify its return value before making any changes to the data store. This method is used to confirm execution of an operation before a change is made to system state, such as clearing content.
`System.Management.Automation.Provider.CmdletProvider.ShouldProcess` sends the name of the resource to be changed to the user, with the Windows PowerShell runtime taking into account any command line settings or preference variables in determining what should be displayed.

After the call to

`System.Management.Automation.Provider.CmdletProvider.ShouldProcess` returns `true`, if potentially dangerous system modifications can be made, the `System.Management.Automation.Provider.IPropertyCmdletProvider.ClearProperty*` method should call the `System.Management.Automation.Provider.CmdletProvider.ShouldContinue` method. This method sends a confirmation message to the user to allow additional feedback to indicate that the potentially dangerous operation should be continued.

Attaching Dynamic Parameters to the Clear-ItemProperty Cmdlet

The `Clear-ItemProperty` cmdlet might require additional parameters that are specified dynamically at runtime. To provide these dynamic parameters, the Windows PowerShell property provider must implement the

`System.Management.Automation.Provider.IPropertyCmdletProvider.ClearPropertyDynamicParameters*` method. This method returns an object that has properties and fields with parsing attributes similar to a cmdlet class or a

`System.Management.Automation.RuntimeDefinedParameterDictionary` object. The `null` value can be returned if no dynamic parameters are to be added.

Here is the default implementation of

`System.Management.Automation.Provider.IPropertyCmdletProvider.ClearPropertyDynamicParameters*` from the `TemplateProvider.cs` file provided by Windows PowerShell.

Building the Windows PowerShell provider

See [How to Register Cmdlets, Providers, and Host Applications](#).

See Also

[Windows PowerShell provider](#)

[Design Your Windows PowerShell provider](#)

[Extending Object Types and Formatting](#)

[How to Register Cmdlets, Providers, and Host Applications](#)

Windows PowerShell Programmer's Guide

Article • 09/17/2021

This programmer's guide is targeted at developers who are interested in providing a command-line management environment for system administrators. Windows PowerShell provides a simple way for you to build management commands that expose .NET objects, while allowing Windows PowerShell to do most of the work for you.

In traditional command development, you are required to write a parameter parser, a parameter binder, filters, and all other functionality exposed by each command. Windows PowerShell provides the following to make it easy for you to write commands:

- A powerful Windows PowerShell runtime (execution engine) with its own parser and a mechanism for automatically binding command parameters.
- Utilities for formatting and displaying command results using a command line interpreter (CLI).
- Support for high levels of functionality (through Windows PowerShell providers) that make it easy to access stored data.

At little cost, you can represent a .NET object by a rich command or set of commands that will offer a complete command-line experience to the administrator.

The next section covers the key Windows PowerShell concepts and terms. Familiarize yourself with these concepts and terms before starting development.

About Windows PowerShell

Windows PowerShell defines several types of commands that you can use in development. These commands include: functions, filters, scripts, aliases, and executables (applications). The main command type discussed in this guide is a simple, small command called a "cmdlet". Windows PowerShell furnishes a set of cmdlets and fully supports cmdlet customization to suit your environment. The Windows PowerShell runtime processes all command types just as it does cmdlets, using pipelines.

In addition to commands, Windows PowerShell supports various customizable Windows PowerShell providers that make available specific sets of cmdlets. The shell operates within the Windows PowerShell-provided host application (`powershell.exe`), but it is equally accessible from a custom host application that you can develop to meet specific requirements. For more information, see [How Windows PowerShell Works](#).

Windows PowerShell Cmdlets

A cmdlet is a lightweight command that is used in the Windows PowerShell environment. The Windows PowerShell runtime invokes these cmdlets within the context of automation scripts that are provided at the command line, and the Windows PowerShell runtime also invokes them programmatically through Windows PowerShell APIs.

For more information about cmdlets, see [Writing a Windows PowerShell Cmdlet](#).

Windows PowerShell Providers

In performing administrative tasks, the user may need to examine data stored in a data store (for example, the file system, the Windows Registry, or a certificate store). To make these operations easier, Windows PowerShell defines a module called a Windows PowerShell provider that can be used to access a specific data store, such as the Windows Registry. Each provider supports a set of related cmdlets to give the user a symmetrical view of the data in the store.

Windows PowerShell provides several default Windows PowerShell providers. For example, the Registry provider supports navigation and manipulation of the Windows Registry. Registry keys are represented as items, and registry values are treated as properties.

If you expose a data store that the user will need to access, you might need to write your own Windows PowerShell provider, as described in [Creating Windows PowerShell Providers](#). For more information about Windows PowerShell providers, see [How Windows PowerShell Works](#).

Host Application

Windows PowerShell includes the default host application powershell.exe, which is a console application that interacts with the user and hosts the Windows PowerShell runtime using a console window.

Only rarely will you need to write your own host application for Windows PowerShell, although customization is supported. One case in which you might need your own application is when you have a requirement for a GUI interface that is richer than the interface provided by the default host application. You might also want a custom application when you are basing your GUI on the command line. For more information, see [How to Create a Windows PowerShell Host Application](#).

Windows PowerShell Runtime

The Windows PowerShell runtime is the execution engine that implements command processing. It includes the classes that provide the interface between the host application and Windows PowerShell commands and providers. The Windows PowerShell runtime is implemented as a runspace object for the current Windows PowerShell session, which is the operational environment in which the shell and the commands execute. For operational details, see [How Windows PowerShell Works](#).

Windows PowerShell Language

The Windows PowerShell language provides scripting functions and mechanisms to invoke commands. For complete scripting information, see the Windows PowerShell Language Reference shipped with Windows PowerShell.

Extended Type System (ETS)

Windows PowerShell provides access to a variety of different objects, such as .NET and XML objects. As a consequence, to present a common abstraction for all object types the shell uses its extended type system (ETS). Most ETS functionality is transparent to the user, but the script or .NET developer uses it for the following purposes:

- Viewing a subset of the members of specific objects. Windows PowerShell provides an "adapted" view of several specific object types.
- Adding members to existing objects.
- Access to serialized objects.
- Writing customized objects.

Using ETS, you can create flexible new "types" that are compatible with the Windows PowerShell language. If you are a .NET developer, you are able to work with objects using the same semantics as the Windows PowerShell language applies to scripting, for example, to determine if an object evaluates to `true`.

For more information about ETS and how Windows PowerShell uses objects, see [Windows PowerShell Object Concepts](#).

Programming for Windows PowerShell

Windows PowerShell defines its code for commands, providers, and other program modules using the .NET Framework. You are not confined to the use of Microsoft Visual Studio in creating customized modules for Windows PowerShell, although the samples provided in this

guide are known to run in this tool. You can use any .NET language that supports class inheritance and the use of attributes. In some cases, Windows PowerShell APIs require the programming language to be able to access generic types.

Programmer's Reference

For reference when developing for Windows PowerShell, see the [Windows PowerShell SDK](#).

Getting Started Using Windows PowerShell

For more information about starting to use the Windows PowerShell shell, see the [Getting Started with Windows PowerShell](#) shipped with Windows PowerShell. A Quick Reference tri-fold document is also supplied as a primer for cmdlet use.

Contents of This Guide

 Expand table

Topic	Definition
How to Create a Windows PowerShell Provider	This section describes how to build a Windows PowerShell provider for Windows PowerShell.
How to Create a Windows PowerShell Host Application	This section describes how to write a host application that manipulates a runspace and how to write a host application that implements its own custom host.
How to Create a Windows PowerShell Snap-in	This section describes how to create a snap-in that is used to register all cmdlets and providers in an assembly and how to create a custom snap-in.
How to Create a Console Shell	This section describes how to create a console shell that is not extensible.
Windows PowerShell Concepts	This section contains conceptual information that will help you understand Windows PowerShell from the viewpoint of a developer.

See Also

[Windows PowerShell SDK](#)

Windows PowerShell Concepts

Article • 10/22/2021

This section contains conceptual information that will help you understand PowerShell from a developer's viewpoint.

[] Expand table

Topic Name	Description
about_Objects	Description of PowerShell objects. For more information, see About Object Creation
Creating Runspaces	The operating environments where commands are processed. For more information, see Runspace Class .
Extending Output Objects	How to extend PowerShell objects. For more information, see About Types.ps1xml
Registering Cmdlets	How to make modules and snap-ins available in PowerShell. For more information, see Modules and Snap-ins .
Requesting Confirmation from Cmdlets	How cmdlets and providers request feedback from the user before an action is taken.
RuntimeDefinedParameter Class	Runtime parameter declarations.
System.Management.Automation Namespace	Overview of PowerShell API namespaces.
Windows PowerShell Provider Overview	Overview about PowerShell providers that are used to access data stores.
Writing Help for PowerShell Cmdlets	How to write PowerShell cmdlet Help.

See also

[PowerShell Class](#)

[PowerShell API Reference](#)

[Windows PowerShell Programmer's Guide](#)

[Writing Help for Windows PowerShell Modules](#)

[Writing a Windows PowerShell Provider](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Windows PowerShell Sample Code

Article • 09/15/2023

Windows PowerShell samples are available through the Windows SDK. This section contains the sample code that is contained in the Windows SDK samples.

ⓘ Note

When the Windows SDK is installed, a **Samples** directory is created in which all the Windows PowerShell samples are made available. A typical installation directory is **C:\Program Files\Microsoft SDKs\Windows\v6.0**. Start Windows PowerShell and type "`cd Samples\SysMgmt\PowerShell`" to locate the samples' directory. In this document, the Windows PowerShell Samples directory is referred to as **<PowerShell Samples>**.

Sample Code Listing

[+] Expand table

Sample Code	Description
AccessDbProviderSample01 Code Sample	This is the provider described in Creating a Basic Windows PowerShell Provider .
AccessDbProviderSample02 Code Sample	This is the provider described in Creating a Windows PowerShell Drive Provider .
AccessDbProviderSample03 Code Sample	This is the provider described in Creating a Windows PowerShell Item Provider .
AccessDbProviderSample04 Code Sample	This is the provider described in Creating a Windows PowerShell Container Provider .
AccessDbProviderSample05 Code Sample	This is the provider described in Creating a Windows PowerShell Navigation Provider .
AccessDbProviderSample06 Code Sample	This is the provider described in Creating a Windows PowerShell Content Provider .
GetProc01 Code Samples	This is the basic <code>Get-Process</code> cmdlet sample described in Creating Your First Cmdlet .
GetProc02 Code Samples	This is the <code>Get-Process</code> cmdlet sample described in Adding Parameters that Process Command-Line Input .

Sample Code	Description
GetProc03 Code Samples	This is the <code>Get-Process</code> cmdlet sample described in Adding Parameters that Process Pipeline Input .
GetProc04 Code Samples	This is the <code>Get-Process</code> cmdlet sample described in Adding Non-terminating Error Reporting to Your Cmdlet .
GetProc05 Code Samples	This <code>Get-Process</code> cmdlet is similar to the cmdlet described in Adding Non-terminating Error Reporting to Your Cmdlet .
StopProc01 Code Samples	This is the <code>Stop-Process</code> cmdlet sample described in Creating a Cmdlet That Modifies the System .
StopProcessSample04 Code Samples	This is the <code>Stop-Process</code> cmdlet sample described in Adding Parameter Sets to a Cmdlet .
Runspace01 Code Samples	These are the code samples for the runspace described in Creating a Console Application That Runs a Specified Command .
Runspace02 Code Samples	This sample uses the <code>System.Management.Automation.RunspaceInvoke</code> class to execute the <code>Get-Process</code> cmdlet synchronously.
RunSpace03 Code Samples	These are the code samples for the runspace described in "Creating a Console Application That Runs a Specified Script".
RunSpace04 Code Samples	This is a code sample for a runspace that uses the <code>System.Management.Automation.RunspaceInvoke</code> class to execute a script that generates a terminating error.
RunSpace05 Code Sample	
RunSpace06 Code Sample	
RunSpace07 Code Sample	
RunSpace08 Code Sample	
RunSpace09 Code Sample	
RunSpace10 Code Sample	This is the source code for the Runspace10 sample, which adds a cmdlet to <code>System.Management.Automation.Runspaces.RunspaceConfiguration</code> and then uses the modified configuration information to create the runspace.

See Also

- [Windows PowerShell Programmer's Guide](#)

- Windows PowerShell SDK

AccessDbProviderSample01 Code Sample

Article • 09/17/2021

The following code shows the implementation of the Windows PowerShell provider described in [Creating a Basic Windows PowerShell Provider](#). This implementation provides methods for starting and stopping the provider, and although it does not provide a means to access a data store or to get or set the data in the data store, it does provide the basic functionality that is required by all providers.

ⓘ Note

You can download the C# source file (AccessDBSampleProvider01.cs) for this provider by using the Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory. For more information about other Windows PowerShell provider implementations, see [Designing Your Windows PowerShell Provider](#).

Code Sample

C#

```
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.ComponentModel;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

        /// <summary>
        /// Simple provider.
        /// </summary>
        [CmdletProvider("AccessDB", ProviderCapabilities.None)]
        public class AccessDBProvider : CmdletProvider
    {

    }
```

```
#endregion AccessDBProvider  
}
```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

AccessDbProviderSample02 Code Sample

Article • 09/17/2021

The following code shows the implementation of the Windows PowerShell provider described in [Creating a Windows PowerShell Drive Provider](#). This implementation creates a path, makes a connection to an Access database, and then removes the drive.

ⓘ Note

You can download the C# source file (AccessDBSampleProvider02.cs) for this provider using the Microsoft Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory. For more information about other Windows PowerShell provider implementations, see [Designing Your Windows PowerShell Provider](#).

Code Sample

C#

```
using System;
using System.IO;
using System.Data;
using System.Data.Odbc;
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.ComponentModel;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

        /// <summary>
        /// A PowerShell Provider which acts upon a access data store.
        /// </summary>
        /// <remarks>
        /// This example only demonstrates the drive overrides
        /// </remarks>
        [CmdletProvider("AccessDB", ProviderCapabilities.None)]
        public class AccessDBProvider : DriveCmdletProvider
    {
        #region Drive Manipulation
```

```

/// <summary>
/// Create a new drive. Create a connection to the database file and
set
/// the Connection property in the PSDriveInfo.
/// </summary>
/// <param name="drive">
/// Information describing the drive to add.
/// </param>
/// <returns>The added drive.</returns>
protected override PSDriveInfo NewDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            null)
        );
    }

    return null;
}

// check if drive root is not null or empty
// and if its an existing file
if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root)
== false))
{
    WriteError(new ErrorRecord(
        new ArgumentException("drive.Root"),
        "NoRoot",
        ErrorCategory.InvalidArgument,
        drive)
    );
}

return null;
}

// create a new drive and create an ODBC connection to the new
drive
AccessDBPSDriveInfo accessDBPSDriveInfo = new
AccessDBPSDriveInfo(drive);

OdbcConnectionStringBuilder builder = new
OdbcConnectionStringBuilder();

builder.Driver = "Microsoft Access Driver (*.mdb)";
builder.Add("DBQ", drive.Root);

OdbcConnection conn = new
OdbcConnection(builder.ConnectionString);
conn.Open();
accessDBPSDriveInfo.Connection = conn;

```

```

        return accessDBPSDriveInfo;
    } // NewDrive

    /// <summary>
    /// Removes a drive from the provider.
    /// </summary>
    /// <param name="drive">The drive to remove.</param>
    /// <returns>The drive removed.</returns>
    protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)
    {
        // check if drive object is null
        if (drive == null)
        {
            WriteError(new ErrorRecord(
                new ArgumentNullException("drive"),
                "NullDrive",
                ErrorCategory.InvalidArgument,
                drive)
            );
        }

        return null;
    }

    // close ODBC connection to the drive
    AccessDBPSDriveInfo accessDBPSDriveInfo = drive as
AccessDBPSDriveInfo;

    if (accessDBPSDriveInfo == null)
    {
        return null;
    }
    accessDBPSDriveInfo.Connection.Close();

    return accessDBPSDriveInfo;
} // RemoveDrive

#endregion Drive Manipulation

} // AccessDBProvider

#endregion AccessDBProvider

#region AccessDBPSDriveInfo

/// <summary>
/// Any state associated with the drive should be held here.
/// In this case, it's the connection to the database.
/// </summary>
internal class AccessDBPSDriveInfo : PSDriveInfo
{
    private OdbcConnection connection;

    /// <summary>
    /// ODBC connection information.

```

```
    /// </summary>
    public OdbcConnection Connection
    {
        get { return connection; }
        set { connection = value; }
    }

    /// <summary>
    /// Constructor that takes one argument
    /// </summary>
    /// <param name="driveInfo">Drive provided by this provider</param>
    public AccessDBPSDriveInfo(PSDriveInfo driveInfo)
        : base(driveInfo)
    { }

} // class AccessDBPSDriveInfo

#endregion AccessDBPSDriveInfo
}
```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

AccessDbProviderSample03 Code Sample

Article • 11/05/2021

The following code shows the implementation of the Windows PowerShell provider described in [Creating a Windows PowerShell Item Provider](#). This provider that can manipulate the data in a data store.

ⓘ Note

You can download the C# source file (AccessDBSampleProvider03.cs) for this provider using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory. For more information about other Windows PowerShell provider implementations, see [Designing Your Windows PowerShell Provider](#).

Code Sample

C#

```
using System;
using System.IO;
using System.Data;
using System.Data.Odbc;
using System.Collections.ObjectModel;
using System.Text;
using System.Diagnostics;
using System.Text.RegularExpressions;
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.ComponentModel;
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

        /// <summary>
        /// A PowerShell Provider which acts upon a access database.
        /// </summary>
        /// <remarks>
        /// This example implements the item overloads.
    #endregion
}
```

```

/// </remarks>
[CmdletProvider("AccessDB", ProviderCapabilities.None)]

public class AccessDBProvider : ItemCmdletProvider
{
    #region Drive Manipulation

    /// <summary>
    /// Create a new drive. Create a connection to the database file and
    set
    /// the Connection property in the PSDriveInfo.
    /// </summary>
    /// <param name="drive">
    /// Information describing the drive to add.
    /// </param>
    /// <returns>The added drive.</returns>
    protected override PSDriveInfo NewDrive(PSDriveInfo drive)
    {
        // check if drive object is null
        if (drive == null)
        {
            WriteError(new ErrorRecord(
                new ArgumentNullException("drive"),
                "NullDrive",
                ErrorCategory.InvalidArgument,
                null)
            );
            return null;
        }

        // check if drive root is not null or empty
        // and if its an existing file
        if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root)
== false))
        {
            WriteError(new ErrorRecord(
                new ArgumentException("drive.Root"),
                "NoRoot",
                ErrorCategory.InvalidArgument,
                drive)
            );
            return null;
        }

        // create a new drive and create an ODBC connection to the new
        drive
        AccessDBPSDriveInfo accessDBPSDriveInfo = new
        AccessDBPSDriveInfo(drive);

        OdbcConnectionStringBuilder builder = new
        OdbcConnectionStringBuilder();

        builder.Driver = "Microsoft Access Driver (*.mdb)";
    }
}

```

```

        builder.Add("DBQ", drive.Root);

        OdbcConnection conn = new
OdbcConnection(builder.ConnectionString);
        conn.Open();
        accessDBPSDriveInfo.Connection = conn;

        return accessDBPSDriveInfo;
    } // NewDrive

    /// <summary>
    /// Removes a drive from the provider.
    /// </summary>
    /// <param name="drive">The drive to remove.</param>
    /// <returns>The drive removed.</returns>
protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            drive)
        );
    }

    return null;
}

// close ODBC connection to the drive
AccessDBPSDriveInfo accessDBPSDriveInfo = drive as
AccessDBPSDriveInfo;

if (accessDBPSDriveInfo == null)
{
    return null;
}
accessDBPSDriveInfo.Connection.Close();

return accessDBPSDriveInfo;
} // RemoveDrive

#endregion Drive Manipulation

#region Item Methods

    /// <summary>
    /// Retrieves an item using the specified path.
    /// </summary>
    /// <param name="path">The path to the item to return.</param>
protected override void GetItem(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))

```

```

    {
        WriteItemObject(this.PSDriveInfo, path, true);
        return;
    } // if (PathIsDrive...

    // Get table name and row information from the path and do
    // necessary actions
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        DatabaseTableInfo table = GetTable(tableName);
        WriteItemObject(table, path, true);
    }
    else if (type == PathType.Row)
    {
        DatabaseRowInfo row = GetRow(tableName, rowNumber);
        WriteItemObject(row, path, false);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }

} // GetItem

/// <summary>
/// Set the content of a row of data specified by the supplied path
/// parameter.
/// </summary>
/// <param name="path">Specifies the path to the row whose columns
/// will be updated.</param>
/// <param name="values">Comma separated string of values</param>
protected override void SetItem(string path, object values)
{
    // Get type, table name and row number from the path specified
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type != PathType.Row)
    {
        WriteError(new ErrorRecord(new NotSupportedException(
            "SetNotSupported"), "", ErrorCategory.InvalidOperation, path));

        return;
    }
}

```

```

// Get in-memory representation of table
OdbcDataAdapter da = GetAdapterForTable(tableName);

if (da == null)
{
    return;
}
DataSet ds = GetDataSetForTable(da, tableName);
DataTable table = GetDataTable(ds, tableName);

if (rowNumber >= table.Rows.Count)
{
    // The specified row number has to be available. If not
    // NewItem has to be used to add a new row
    throw new ArgumentException("Row specified is not
available");
} // if (rowNum...

string[] colValues = (values as string).Split(',');
// set the specified row
DataRow row = table.Rows[rowNumber];

for (int i = 0; i < colValues.Length; i++)
{
    row[i] = colValues[i];
}

// Update the table
if (ShouldProcess(path, "SetItem"))
{
    da.Update(ds, tableName);
}

} // SetItem

/// <summary>
/// Test to see if the specified item exists.
/// </summary>
/// <param name="path">The path to the item to verify.</param>
/// <returns>True if the item is found.</returns>
protected override bool ItemExists(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        return true;
    }

    // Obtain type, table name and row number from path
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

```

```

DatabaseTableInfo table = GetTable(tableName);

    if (type == PathType.Table)
    {
        // if specified path represents a table then
DatabaseTableInfo
        // object for the same should exist
        if (table != null)
        {
            return true;
        }
    }
    else if (type == PathType.Row)
    {
        // if specified path represents a row then DatabaseTableInfo
should
        // exist for the table and then specified row number must be
within
        // the maximum row count in the table
        if (table != null && rowCount < table.RowCount)
        {
            return true;
        }
    }

    return false;
}

} // ItemExists

/// <summary>
/// Test to see if the specified path is syntactically valid.
/// </summary>
/// <param name="path">The path to validate.</param>
/// <returns>True if the specified path is valid.</returns>
protected override bool IsValidPath(string path)
{
    bool result = true;

    // check if the path is null or empty
    if (String.IsNullOrEmpty(path))
    {
        result = false;
    }

    // convert all separators in the path to a uniform one
    path = NormalizePath(path);

    // split the path into individual chunks
    string[] pathChunks = path.Split(pathSeparator.ToCharArray());

    foreach (string pathChunk in pathChunks)
    {
        if (pathChunk.Length == 0)
        {

```

```

        result = false;
    }
}
return result;
} // IsValidPath

#endregion Item Overloads

#region Helper Methods

/// <summary>
/// Checks if a given path is actually a drive name.
/// </summary>
/// <param name="path">The path to check.</param>
/// <returns>
/// True if the path given represents a drive, false otherwise.
/// </returns>
private bool PathIsDrive(string path)
{
    // Remove the drive name and first path separator. If the
    // path is reduced to nothing, it is a drive. Also if its
    // just a drive then there wont be any path separators
    if (String.IsNullOrEmpty(
        path.Replace(this.PSDriveInfo.Root, ""))
        ||
        String.IsNullOrEmpty(
            path.Replace(this.PSDriveInfo.Root + pathSeparator,
            "")))
    {
        return true;
    }
    else
    {
        return false;
    }
} // PathIsDrive

/// <summary>
/// Breaks up the path into individual elements.
/// </summary>
/// <param name="path">The path to split.</param>
/// <returns>An array of path segments.</returns>
private string[] ChunkPath(string path)
{
    // Normalize the path before splitting
    string normalPath = NormalizePath(path);

    // Return the path with the drive name and first path
    // separator character removed, split by the path separator.
    string pathNoDrive = normalPath.Replace(this.PSDriveInfo.Root
        + pathSeparator, "");

    return pathNoDrive.Split(pathSeparator.ToCharArray());
} // ChunkPath

```

```

    ///<summary>
    /// Adapts the path, making sure the correct path separator
    /// character is used.
    ///</summary>
    ///<param name="path"></param>
    ///<returns></returns>
    private string NormalizePath(string path)
    {
        string result = path;

        if (!String.IsNullOrEmpty(path))
        {
            result = path.Replace("/", pathSeparator);
        }

        return result;
    } // NormalizePath

    ///<summary>
    /// Chunks the path and returns the table name and the row number
    /// from the path
    ///</summary>
    ///<param name="path">Path to chunk and obtain information</param>
    ///<param name="tableName">Name of the table as represented in the
    /// path</param>
    ///<param name="rowNumber">Row number obtained from the path</param>
    ///<returns>what the path represents</returns>
    private PathType GetNamesFromPath(string path, out string tableName,
out int rowNumber)
    {
        PathType retVal = PathType.Invalid;
        rowNumber = -1;
        tableName = null;

        // Check if the path specified is a drive
        if (PathIsDrive(path))
        {
            return PathType.Database;
        }

        // chunk the path into parts
        string[] pathChunks = ChunkPath(path);

        switch (pathChunks.Length)
        {
            case 1:
            {
                string name = pathChunks[0];

                if (TableNameIsValid(name))
                {
                    tableName = name;
                    retVal = PathType.Table;
                }
            }
        }
    }
}

```

```

        }

        break;

    case 2:
    {
        string name = pathChunks[0];

        if (TableNameIsValid(name))
        {
            tableName = name;
        }

        int number = SafeConvertRowNumber(pathChunks[1]);

        if (number >= 0)
        {
            rowNum = number;
            retVal = PathType.Row;
        }
        else
        {
            WriteError(new ErrorRecord(
                new ArgumentException("Row number is not
valid"),
                "RowNumberNotValid",
                ErrorCategory.InvalidArgument,
                path));
        }
    }
    break;

    default:
    {
        WriteError(new ErrorRecord(
            new ArgumentException("The path supplied has too
many segments"),
            "PathNotValid",
            ErrorCategory.InvalidArgument,
            path));
    }
    break;
} // switch(pathChunks...

return retVal;
} // GetNamesFromPath

/// <summary>
/// Throws an argument exception stating that the specified path does
/// not represent either a table or a row
/// </summary>
/// <param name="path">path which is invalid</param>
private void ThrowTerminatingInvalidPathException(string path)
{
    StringBuilder message = new StringBuilder("Path must represent
either a table or a row :");

```

```

        message.Append(path);

        throw new ArgumentException(message.ToString());
    }

    /// <summary>
    /// Retrieve the list of tables from the database.
    /// </summary>
    /// <returns>
    /// Collection of DatabaseTableInfo objects, each object representing
    /// information about one database table
    /// </returns>
    private Collection<DatabaseTableInfo> GetTables()
    {
        Collection<DatabaseTableInfo> results =
            new Collection<DatabaseTableInfo>();

        // using ODBC connection to the database and get the schema of
        tables
        AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

        if (di == null)
        {
            return null;
        }

        OdbcConnection connection = di.Connection;
        DataTable dt = connection.GetSchema("Tables");
        int count;

        // iterate through all rows in the schema and create
        DatabaseTableInfo
        // objects which represents a table
        foreach (DataRow dr in dt.Rows)
        {
            String tableName = dr["TABLE_NAME"] as String;
            DataColumnCollection columns = null;

            // find the number of rows in the table
            try
            {
                String cmd = "Select count(*) from \\" + tableName + "\\";
                OdbcCommand command = new OdbcCommand(cmd, connection);

                count = (Int32)command.ExecuteScalar();
            }
            catch
            {
                count = 0;
            }

            // create DatabaseTableInfo object representing the table
            DatabaseTableInfo table =
                new DatabaseTableInfo(dr, tableName, count, columns);
        }
    }
}

```

```

        results.Add(table);
    } // foreach (DataRow...
}

return results;
} // GetTables

/// <summary>
/// Return row information from a specified table.
/// </summary>
/// <param name="tableName">The name of the database table from
/// which to retrieve rows.</param>
/// <returns>Collection of row information objects.</returns>
private Collection<DatabaseRowInfo> GetRows(string tableName)
{
    Collection<DatabaseRowInfo> results =
        new Collection<DatabaseRowInfo>();

    // Obtain rows in the table and add it to the collection
    try
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);

        if (da == null)
        {
            return null;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        int i = 0;
        foreach (DataRow row in table.Rows)
        {
            results.Add(new DatabaseRowInfo(row,
i.ToString(CultureInfo.CurrentCulture)));
            i++;
        } // foreach (DataRow...
    }
    catch (Exception e)
    {
        WriteError(new ErrorRecord(e, "CannotAccessSpecifiedRows",
ErrorCategory.InvalidOperation, tableName));
    }
}

return results;
} // GetRows

/// <summary>
/// Retrieve information about a single table.
/// </summary>
/// <param name="tableName">The table for which to retrieve
/// data.</param>
/// <returns>Table information.</returns>
private DatabaseTableInfo GetTable(string tableName)

```

```

    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            if (String.Equals(tableName, table.Name,
StringComparison.OrdinalIgnoreCase))
            {
                return table;
            }
        }

        return null;
    } // GetTable

    /// <summary>
    /// Obtain a data adapter for the specified Table
    /// </summary>
    /// <param name="tableName">Name of the table to obtain the
    /// adapter for</param>
    /// <returns>Adapter object for the specified table</returns>
    /// <remarks>An adapter serves as a bridge between a DataSet (in
memory
    /// representation of table) and the data source</remarks>
    private OdbcDataAdapter GetAdapterForTable(string tableName)
    {
        OdbcDataAdapter da = null;
        AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

        if (di == null || !TableNameIsValid(tableName) ||
!TableIsPresent(tableName))
        {
            return null;
        }

        OdbcConnection connection = di.Connection;

        try
        {
            // Create a odbc data adpater. This can be sued to update the
            // data source with the records that will be created here
            // using data sets
            string sql = "Select * from " + tableName;
            da = new OdbcDataAdapter(new OdbcCommand(sql, connection));

            // Create a odbc command builder object. This will create sql
            // commands automatically for a single table, thus
            // eliminating the need to create new sql statements for
            // every operation to be done.
            OdbcCommandBuilder cmd = new OdbcCommandBuilder(da);

            // Open the connection if its not already open
            if (connection.State != ConnectionState.Open)
            {
                connection.Open();
            }
        }
    }
}

```

```

        catch (Exception e)
        {
            WriteError(new ErrorRecord(e, "CannotAccessSpecifiedTable",
                ErrorCategory.InvalidOperation, tableName));
        }

        return da;
    } // GetAdapterForTable

    /// <summary>
    /// Gets the DataSet (in memory representation) for the table
    /// for the specified adapter
    /// </summary>
    /// <param name="adapter">Adapter to be used for obtaining
    /// the table</param>
    /// <param name="tableName">Name of the table for which a
    /// DataSet is required</param>
    /// <returns>The DataSet with the filled in schema</returns>
    private DataSet GetDataSetForTable(OdbcDataAdapter adapter, string
tableName)
{
    Debug.Assert(adapter != null);

    // Create a dataset object which will provide an in-memory
    // representation of the data being worked upon in the
    // data source.
    DataSet ds = new DataSet();

    // Create a table named "Table" which will contain the same
    // schema as in the data source.
    //adapter.FillSchema(ds, SchemaType.Source);
    adapter.Fill(ds, tableName);
    ds.Locale = CultureInfo.InvariantCulture;

    return ds;
} //GetDataSetForTable

    /// <summary>
    /// Get the DataTable object which can be used to operate on
    /// for the specified table in the data source
    /// </summary>
    /// <param name="ds">DataSet object which contains the tables
    /// schema</param>
    /// <param name="tableName">Name of the table</param>
    /// <returns>Corresponding DataTable object representing
    /// the table</returns>
    ///
private DataTable GetDataTable(DataSet ds, string tableName)
{
    Debug.Assert(ds != null);
    Debug.Assert(tableName != null);

    DataTable table = ds.Tables[tableName];
    table.Locale = CultureInfo.InvariantCulture;
}

```

```

        return table;
    } // GetDataTable

    /// <summary>
    /// Retrieves a single row from the named table.
    /// </summary>
    /// <param name="tableName">The table that contains the
    /// numbered row.</param>
    /// <param name="row">The index of the row to return.</param>
    /// <returns>The specified table row.</returns>
    private DatabaseRowInfo GetRow(string tableName, int row)
    {
        Collection<DatabaseRowInfo> di = GetRows(tableName);

        // if the row is invalid write an appropriate error else return
        the
        // corresponding row information
        if (row < di.Count && row >= 0)
        {
            return di[row];
        }
        else
        {
            WriteError(new ErrorRecord(
                new ItemNotFoundException(),
                "RowNotFound",
                ErrorCategory.ObjectNotFound,
                row.ToString(CultureInfo.CurrentCulture))
            );
        }

        return null;
    } // GetRow

    /// <summary>
    /// Method to safely convert a string representation of a row number
    /// into its Int32 equivalent
    /// </summary>
    /// <param name="rowNumberAsStr">String representation of the row
    /// number</param>
    /// <remarks>If there is an exception, -1 is returned</remarks>
    private int SafeConvertRowNumber(string rowNumberAsStr)
    {
        int rowNumber = -1;
        try
        {
            rowNumber = Convert.ToInt32(rowNumberAsStr,
CultureInfo.CurrentCulture);
        }
        catch (FormatException fe)
        {
            WriteError(new ErrorRecord(fe, "RowStringFormatNotValid",
                ErrorCategory.InvalidData, rowNumberAsStr));
        }
        catch (OverflowException oe)

```

```

    {
        WriteError(new ErrorRecord(oe,
"RowStringConversionToNumberFailed",
            ErrorCategory.InvalidData, rowNumberAsStr));
    }

    return rowNumber;
} // SafeConvertRowNumber

/// <summary>
/// Check if a table name is valid
/// </summary>
/// <param name="tableName">Table name to validate</param>
/// <remarks>Helps to check for SQL injection attacks</remarks>
private bool TableNameIsValid(string tableName)
{
    Regex exp = new Regex(pattern, RegexOptions.Compiled |
RegexOptions.IgnoreCase);

    if (exp.IsMatch(tableName))
    {
        return true;
    }
    WriteError(new ErrorRecord(
        new ArgumentException("Table name not valid"),
"TableNameNotValid",
            ErrorCategory.InvalidArgument, tableName));
    return false;
} // TableNameIsValid

/// <summary>
/// Checks to see if the specified table is present in the
/// database
/// </summary>
/// <param name="tableName">Name of the table to check</param>
/// <returns>true, if table is present, false otherwise</returns>
private bool TableIsPresent(string tableName)
{
    // using ODBC connection to the database and get the schema of
tables
    AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;
    if (di == null)
    {
        return false;
    }

    OdbcConnection connection = di.Connection;
    DataTable dt = connection.GetSchema("Tables");

    // check if the specified tableName is available
    // in the list of tables present in the database
    foreach (DataRow dr in dt.Rows)
    {
        string name = dr["TABLE_NAME"] as string;
        if (name.Equals(tableName,

```

```

        StringComparison.OrdinalIgnoreCase))
    {
        return true;
    }
}

WriteError(new ErrorRecord(
    new ArgumentException("Specified Table is not present in
database"), "TableNotAvailable",
    ErrorCategory.InvalidArgument, tableName));

return false;
}// TableIsPresent

#endregion Helper Methods

#region Private Properties

private string pathSeparator = "\\";
private static string pattern = @"^+[a-z]+[0-9]*_*$";

private enum PathType { Database, Table, Row, Invalid };

#endregion Private Properties
}

#endregion AccessDBProvider

#region Helper Classes

#region AccessDBPSDriveInfo

/// <summary>
/// Any state associated with the drive should be held here.
/// In this case, it's the connection to the database.
/// </summary>
internal class AccessDBPSDriveInfo : PSDriveInfo
{
    private OdbcConnection connection;

    /// <summary>
    /// ODBC connection information.
    /// </summary>
    public OdbcConnection Connection
    {
        get { return connection; }
        set { connection = value; }
    }

    /// <summary>
    /// Constructor that takes one argument
    /// </summary>
    /// <param name="driveInfo">Drive provided by this provider</param>
    public AccessDBPSDriveInfo(PSDriveInfo driveInfo)
        : base(driveInfo)

```

```
{ }

} // class AccessDBPSDriveInfo

#endregion AccessDBPSDriveInfo

#region DatabaseTableInfo

/// <summary>
/// Contains information specific to the database table.
/// Similar to the DirectoryInfo class.
/// </summary>
public class DatabaseTableInfo
{
    /// <summary>
    /// Row from the "tables" schema
    /// </summary>
    public DataRow Data
    {
        get
        {
            return data;
        }
        set
        {
            data = value;
        }
    }
    private DataRow data;

    /// <summary>
    /// The table name.
    /// </summary>
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
    private String name;

    /// <summary>
    /// The number of rows in the table.
    /// </summary>
    public int RowCount
    {
        get
        {
            return rowCount;
        }
    }
}
```

```

        set
    {
        rowCount = value;
    }
}
private int rowCount;

/// <summary>
/// The column definitions for the table.
/// </summary>
public DataColumnCollection Columns
{
    get
    {
        return columns;
    }
    set
    {
        columns = value;
    }
}
private DataColumnCollection columns;

/// <summary>
/// Constructor.
/// </summary>
/// <param name="row">The row definition.</param>
/// <param name="name">The table name.</param>
/// <param name="rowCount">The number of rows in the table.</param>
/// <param name="columns">Information on the column tables.</param>
public DatabaseTableInfo(DataRow row, string name, int rowCount,
                         DataColumnCollection columns)
{
    Name = name;
    Data = row;
    RowCount = rowCount;
    Columns = columns;
} // DatabaseTableInfo
} // class DatabaseTableInfo

#endregion DatabaseTableInfo

#region DatabaseRowInfo

/// <summary>
/// Contains information specific to an individual table row.
/// Analogous to the FileInfo class.
/// </summary>
public class DatabaseRowInfo
{
    /// <summary>
    /// Row data information.
    /// </summary>
    public DataRow Data
    {

```

```

        get
    {
        return data;
    }
    set
    {
        data = value;
    }
}
private DataRow data;

/// <summary>
/// The row index.
/// </summary>
public string RowNumber
{
    get
    {
        return RowNumber;
    }
    set
    {
        RowNumber = value;
    }
}
private string RowNumber;

/// <summary>
/// Constructor.
/// </summary>
/// <param name="row">The row information.</param>
/// <param name="name">The row index.</param>
public DatabaseRowInfo(DataRow row, string name)
{
    RowNumber = name;
    Data = row;
} // DatabaseRowInfo
} // class DatabaseRowInfo

#endregion DatabaseRowInfo

#region Helper Classes
}

```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

AccessDbProviderSample04 Code Sample

Article • 09/17/2021

The following code shows the implementation of the Windows PowerShell provider described in [Creating a Windows PowerShell Container Provider](#). This provider works on multi-layer data stores. For this type of data store, the top level of the store contains the root items and each subsequent level is referred to as a node of child items. By allowing the user to work on these child nodes, a user can interact hierarchically through the data store.

Code Sample

C#

```
using System;
using System.IO;
using System.Data;
using System.Data.Odbc;
using System.Data.OleDb;
using System.Diagnostics;
using System.Collections.ObjectModel;
using System.Text;
using System.Text.RegularExpressions;
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.ComponentModel;
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

        /// <summary>
        /// A PowerShell Provider which acts upon an Access database
        /// </summary>
        /// <remarks>
        /// This example implements the container overloads</remarks>
        [CmdletProvider("AccessDB", ProviderCapabilities.None)]
        public class AccessDBProvider : ContainerCmdletProvider
    {

        #region Drive Manipulation

            /// <summary>
            /// Create a new drive. Create a connection to the database file and
            set
```

```

///> the Connection property in the PSDriveInfo.
///</summary>
///<param name="drive">
/// Information describing the drive to add.
///</param>
///<returns>The added drive.</returns>
protected override PSDriveInfo NewDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            null)
        );
    }

    return null;
}

// check if drive root is not null or empty
// and if its an existing file
if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root)
== false))
{
    WriteError(new ErrorRecord(
        new ArgumentException("drive.Root"),
        "NoRoot",
        ErrorCategory.InvalidArgument,
        drive)
    );
}

return null;
}

// create a new drive and create an ODBC connection to the new
drive
AccessDBPSDriveInfo accessDBPSDriveInfo = new
AccessDBPSDriveInfo(drive);

OdbcConnectionStringBuilder builder = new
OdbcConnectionStringBuilder();

builder.Driver = "Microsoft Access Driver (*.mdb)";
builder.Add("DBQ", drive.Root);

OdbcConnection conn = new
OdbcConnection(builder.ConnectionString);
conn.Open();
accessDBPSDriveInfo.Connection = conn;

return accessDBPSDriveInfo;
} // NewDrive

```

```

/// <summary>
/// Removes a drive from the provider.
/// </summary>
/// <param name="drive">The drive to remove.</param>
/// <returns>The drive removed.</returns>
protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            drive)
        );
    }

    return null;
}

// close ODBC connection to the drive
AccessDBPSDriveInfo accessDBPSDriveInfo = drive as
AccessDBPSDriveInfo;

if (accessDBPSDriveInfo == null)
{
    return null;
}
accessDBPSDriveInfo.Connection.Close();

return accessDBPSDriveInfo;
} // RemoveDrive

#endregion Drive Manipulation

#region Item Methods

/// <summary>
/// Retrieves an item using the specified path.
/// </summary>
/// <param name="path">The path to the item to return.</param>
protected override void GetItem(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        WriteItemObject(this.PSDriveInfo, path, true);
        return;
    } // if (PathIsDrive...

    // Get table name and row information from the path and do
    // necessary actions
    string tableName;
    int lineNumber;
}

```

```

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Table)
        {
            DatabaseTableInfo table = GetTable(tableName);
            WriteItemObject(table, path, true);
        }
        else if (type == PathType.Row)
        {
            DatabaseRowInfo row = GetRow(tableName, rowNumber);
            WriteItemObject(row, path, false);
        }
        else
        {
            ThrowTerminatingInvalidOperationException(path);
        }

    } // GetItem

    /// <summary>
    /// Set the content of a row of data specified by the supplied path
    /// parameter.
    /// </summary>
    /// <param name="path">Specifies the path to the row whose columns
    /// will be updated.</param>
    /// <param name="values">Comma separated string of values</param>
protected override void SetItem(string path, object values)
{
    // Get type, table name and row number from the path specified
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type != PathType.Row)
    {
        WriteError(new ErrorRecord(new NotSupportedException(
            "SetNotSupportedException"), "", 
        ErrorCategory.InvalidOperation, path));

        return;
    }

    // Get in-memory representation of table
    OdbcDataAdapter da = GetAdapterForTable(tableName);

    if (da == null)
    {
        return;
    }
    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);
}

```

```

    if (rowNumber >= table.Rows.Count)
    {
        // The specified row number has to be available. If not
        // NewItem has to be used to add a new row
        throw new ArgumentException("Row specified is not
available");
    } // if (rowNum...

    string[] colValues = (values as string).Split(',');
}

// set the specified row
DataRow row = table.Rows[rowNumber];

for (int i = 0; i < colValues.Length; i++)
{
    row[i] = colValues[i];
}

// Update the table
if (ShouldProcess(path, "SetItem"))
{
    da.Update(ds, tableName);
}

} // SetItem

/// <summary>
/// Test to see if the specified item exists.
/// </summary>
/// <param name="path">The path to the item to verify.</param>
/// <returns>True if the item is found.</returns>
protected override bool ItemExists(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        return true;
    }

    // Obtain type, table name and row number from path
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    DatabaseTableInfo table = GetTable(tableName);

    if (type == PathType.Table)
    {
        // if specified path represents a table then
DatabaseTableInfo
            // object for the same should exist
            if (table != null)

```

```

        {
            return true;
        }
    }
    else if (type == PathType.Row)
    {
        // if specified path represents a row then DatabaseTableInfo
should
        // exist for the table and then specified row number must be
within
        // the maximum row count in the table
        if (table != null && rowNum < table.RowCount)
        {
            return true;
        }
    }

    return false;
}

} // ItemExists

/// <summary>
/// Test to see if the specified path is syntactically valid.
/// </summary>
/// <param name="path">The path to validate.</param>
/// <returns>True if the specified path is valid.</returns>
protected override bool IsValidPath(string path)
{
    bool result = true;

    // check if the path is null or empty
    if (String.IsNullOrEmpty(path))
    {
        result = false;
    }

    // convert all separators in the path to a uniform one
    path = NormalizePath(path);

    // split the path into individual chunks
    string[] pathChunks = path.Split(pathSeparator.ToCharArray());

    foreach (string pathChunk in pathChunks)
    {
        if (pathChunk.Length == 0)
        {
            result = false;
        }
    }
    return result;
} // IsValidPath

#endregion Item Overloads

#region Container Overloads
```

```

/// <summary>
/// Return either the tables in the database or the datarows
/// </summary>
/// <param name="path">The path to the parent</param>
/// <param name="recurse">True to return all child items recursively.
/// </param>
protected override void GetChildItems(string path, bool recurse)
{
    // If path represented is a drive then the children in the path
are
    // tables. Hence all tables in the drive represented will have to
be
    // returned
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table, path, true);

            // if the specified item exists and recurse has been set
then
            // all child items within it have to be obtained as well
            if (ItemExists(path) && recurse)
            {
                GetChildItems(path + pathSeparator + table.Name,
recurse);
            }
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
else
{
    // Get the table name, row number and type of path from the
    // path specified
    string tableName;
    int rowCount;

    PathType type = GetNamesFromPath(path, out tableName, out
rowCount);

    if (type == PathType.Table)
    {
        // Obtain all the rows within the table
        foreach (DatabaseRowInfo row in GetRows(tableName))
        {
            WriteItemObject(row, path + pathSeparator +
row.RowNumber,
                            false);
        } // foreach (DatabaseRowInfo...
    }
    else if (type == PathType.Row)
    {
        // In this case the user has directly specified a row,
hence
        // just give that particular row
    }
}

```

```

        DatabaseRowInfo row = GetRow(tableName, rowNum);
        WriteItemObject(row, path + pathSeparator +
row.RowNumber,
                           false);
    }
    else
    {
        // In this case, the path specified is not valid
        ThrowTerminatingInvalidPathException(path);
    }
} // else
} // GetChildItems

/// <summary>
/// Return the names of all child items.
/// </summary>
/// <param name="path">The root path.</param>
/// <param name="returnContainers">Not used.</param>
protected override void GetChildNames(string path,
                                      ReturnContainers returnContainers)
{
    // If the path represented is a drive, then the child items are
    // tables. get the names of all the tables in the drive.
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table.Name, path, true);
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get type, table name and row number from path specified
        string tableName;
        int rowNum;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNum);

        if (type == PathType.Table)
        {
            // Get all the rows in the table and then write out the
            // row numbers.
            foreach (DatabaseRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row.RowNumber, path, false);
            } // foreach (DatabaseRowInfo...
        }
        else if (type == PathType.Row)
        {
            // In this case the user has directly specified a row,
            hence
            // just give that particular row
            DatabaseRowInfo row = GetRow(tableName, rowNum);
        }
    }
}

```

```

                WriteItemObject(row.RowNumber, path, false);
            }
            else
            {
                ThrowTerminatingInvalidOperationException(path);
            }
        } // else
    } // GetChildNames

    /// <summary>
    /// Determines if the specified path has child items.
    /// </summary>
    /// <param name="path">The path to examine.</param>
    /// <returns>
    /// True if the specified path has child items.
    /// </returns>
    protected override bool HasChildItems(string path)
    {
        if (PathIsDrive(path))
        {
            return true;
        }

        return (ChunkPath(path).Length == 1);
    } // HasChildItems

    /// <summary>
    /// Creates a new item at the specified path.
    /// </summary>
    ///
    /// <param name="path">
    /// The path to the new item.
    /// </param>
    ///
    /// <param name="type">
    /// Type for the object to create. "Table" for creating a new table
and
    /// "Row" for creating a new row in a table.
    /// </param>
    ///
    /// <param name="newValue">
    /// Object for creating new instance of a type at the specified path.
For
    /// creating a "Table" the object parameter is ignored and for
creating
    /// a "Row" the object must be of type string which will contain
comma
    /// separated values of the rows to insert.
    /// </param>
    protected override void NewItem(string path, string type,
                                object newValue)
    {
        string tableName;
        int rowNum;
    }
}

```

```

        PathType pt = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (pt == PathType.Invalid)
        {
            ThrowTerminatingInvalidPathException(path);
        }

        // Check if type is either "table" or "row", if not throw an
        // exception
        if (!String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase)
            && !String.Equals(type, "row",
StringComparison.OrdinalIgnoreCase))
        {
            WriteError(new ErrorRecord
                (new ArgumentException("Type must be either
a table or row"),
                    "CannotCreateSpecifiedObject",
                    ErrorCategory.InvalidArgument,
                    path
                )
            );
        }

        throw new ArgumentException("This provider can only create
items of type \"table\" or \"row\"");
    }

    // Path type is the type of path of the container. So if a drive
    // is specified, then a table can be created under it and if a
table
    // is specified, then a row can be created under it. For the sake
of
    // completeness, if a row is specified, then if the row specified
by
    // the path does not exist, a new row is created. However, the
row
    // number may not match as the row numbers only get incremented
based
    // on the number of rows

    if (PathIsDrive(path))
    {
        if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
        {
            // Execute command using ODBC connection to create a
table
            try
            {
                // create the table using an sql statement
                string newTableName = newItemValue.ToString();

                if (!TableNameIsValid(newTableName))
                {

```

```

        return;
    }
    string sql = "create table " + newTableName
        + " (ID INT)";

    // Create the table using the Odbc connection from
the
    // drive.
    AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

    if (di == null)
    {
        return;
    }
    OdbcConnection connection = di.Connection;

    if (ShouldProcess(newTableName, "create"))
    {
        OdbcCommand cmd = new OdbcCommand(sql,
connection);
        cmd.ExecuteScalar();
    }
    catch (Exception ex)
    {
        WriteError(new ErrorRecord(ex,
"CannotCreateSpecifiedTable",
ErrorCategory.InvalidOperation, path)
);
    }
} // if (String...
else if (String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
{
    throw new
        ArgumentException("A row cannot be created under a
database, specify a path that represents a Table");
}
}// if (PathIsDrive...
else
{
    if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
    {
        if (rowNumber < 0)
        {
            throw new
                ArgumentException("A table cannot be created
within another table, specify a path that represents a database");
        }
        else
        {
            throw new
                ArgumentException("A table cannot be created

```

```

inside a row, specify a path that represents a database");
        }
    } //if (String.Equals....
    // if path specified is a row, create a new row
    else if (String.Equals(type, "row",
StringComparison.OrdinalIgnoreCase))
{
    // The user is required to specify the values to be
inserted
    // into the table in a single string separated by commas
    string value = newItemValue as string;

    if (String.IsNullOrEmpty(value))
{
    throw new
        ArgumentException("Value argument must have comma
separated values of each column in a row");
}
    string[] rowValues = value.Split(',');
}

OdbcDataAdapter da = GetAdapterForTable(tableName);

if (da == null)
{
    return;
}

DataSet ds = GetDataSetForTable(da, tableName);
DataTable table = GetDataTable(ds, tableName);

if (rowValues.Length != table.Columns.Count)
{
    string message =
        String.Format(CultureInfo.CurrentCulture,
                    "The table has {0} columns and
the value specified must have so many comma separated values",
                    table.Columns.Count);

    throw new ArgumentException(message);
}

if (!Force && (rowNumber >=0 && rowNumber <
table.Rows.Count))
{
    string message =
String.Format(CultureInfo.CurrentCulture,
                    "The row {0} already
exists. To create a new row specify row number as {1}, or specify path to a
table, or use the -Force parameter",
                    rowNumber,
table.Rows.Count);

    throw new ArgumentException(message);
}

```

```

        if (rowNumber > table.Rows.Count)
        {
            string message =
String.Format(CultureInfo.CurrentCulture,
                           "To create a new row specify row
number as {0}, or specify path to a table",
                           table.Rows.Count);

            throw new ArgumentException(message);
        }

        // Create a new row and update the row with the input
        // provided by the user
        DataRow row = table.NewRow();
        for (int i = 0; i < rowValues.Length; i++)
        {
            row[i] = rowValues[i];
        }
        table.Rows.Add(row);

        if (ShouldProcess(tableName, "update rows"))
        {
            // Update the table from memory back to the data
source
            da.Update(ds, tableName);
        }

        } // else if (String...
    } // else ...

} // NewItem

/// <summary>
/// Copies an item at the specified path to the location specified
/// </summary>
///
/// <param name="path">
/// Path of the item to copy
/// </param>
///
/// <param name="copyPath">
/// Path of the item to copy to
/// </param>
///
/// <param name="recurse">
/// Tells the provider to recurse subcontainers when copying
/// </param>
///
protected override void CopyItem(string path, string copyPath, bool
recurse)
{
    string tableName, copyTableName;
    int rowCount, copyRowCount;

    PathType type = GetNamesFromPath(path, out tableName, out

```

```

rowNumber);

    PathType copyType = GetNamesFromPath(copyPath, out copyTableName,
out copyRowNumber);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(copyPath);
    }

    // Get the table and the table to copy to
    OdbcDataAdapter da = GetAdapterForTable(tableName);
    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    OdbcDataAdapter cda = GetAdapterForTable(copyTableName);
    if (cda == null)
    {
        return;
    }

    DataSet cds = GetDataSetForTable(cda, copyTableName);
    DataTable copyTable = GetDataTable(cds, copyTableName);

    // if source represents a table
    if (type == PathType.Table)
    {
        // if copyPath does not represent a table
        if (copyType != PathType.Table)
        {
            ArgumentException e = new ArgumentException("Table can
only be copied on to another table location");

            WriteError(new ErrorRecord(e, "PathNotValid",
                ErrorCategory.InvalidArgument, copyPath));

            throw e;
        }

        // if table already exists then force parameter should be set
        // to force a copy
        if (!Force && GetTable(copyTableName) != null)
        {
            throw new ArgumentException("Specified path already
exists");
        }
    }
}

```

```

        for (int i = 0; i < table.Rows.Count; i++)
        {
            DataRow row = table.Rows[i];
            DataRow copyRow = copyTable.NewRow();

            copyRow.ItemArray = row.ItemArray;
            copyTable.Rows.Add(copyRow);
        }
    } // if (type == ...
// if source represents a row
else
{
    if (copyType == PathType.Row)
    {
        if (!Force && (copyRowNumber < copyTable.Rows.Count))
        {
            throw new ArgumentException("Specified path already
exists.");
        }

        DataRow row = table.Rows[rowNumber];
        DataRow copyRow = null;

        if (copyRowNumber < copyTable.Rows.Count)
        {
            // copy to an existing row
            copyRow = copyTable.Rows[copyRowNumber];
            copyRow.ItemArray = row.ItemArray;
            copyRow[0] = GetNextID(copyTable);
        }
        else if (copyRowNumber == copyTable.Rows.Count)
        {
            // copy to the next row in the table that will
            // be created
            copyRow = copyTable.NewRow();
            copyRow.ItemArray = row.ItemArray;
            copyRow[0] = GetNextID(copyTable);
            copyTable.Rows.Add(copyRow);
        }
        else
        {
            // attempting to copy to a nonexistent row or a row
            // that cannot be created now - throw an exception
            string message =
String.Format(CultureInfo.CurrentCulture,
                         "The item cannot be specified
to the copied row. Specify row number as {0}, or specify a path to the
table.",

                         table.Rows.Count);

            throw new ArgumentException(message);
        }
    }
    else

```

```

        {
            // destination path specified represents a table,
            // create a new row and copy the item
            DataRow copyRow = copyTable.NewRow();
            copyRow.ItemArray = table.Rows[rowNumber].ItemArray;
            copyRow[0] = GetNextID(copyTable);
            copyTable.Rows.Add(copyRow);
        }
    }

    if (ShouldProcess(copyTableName, "CopyItems"))
    {
        cda.Update(cds, copyTableName);
    }

} //CopyItem

/// <summary>
/// Removes (deletes) the item at the specified path
/// </summary>
///
/// <param name="path">
/// The path to the item to remove.
/// </param>
///
/// <param name="recurse">
/// True if all children in a subtree should be removed, false if
only
/// the item at the specified path should be removed. Is applicable
/// only for container (table) items. Its ignored otherwise (even if
/// specified).
/// </param>
///
/// <remarks>
/// There are no elements in this store which are hidden from the
user.
/// Hence this method will not check for the presence of the Force
/// parameter
/// </remarks>
///

protected override void RemoveItem(string path, bool recurse)
{
    string tableName;
    int rowNumber = 0;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        // if recurse flag has been specified, delete all the rows as
well
        if (recurse)
        {
            OdbcDataAdapter da = GetAdapterForTable(tableName);

```

```

    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    for (int i = 0; i < table.Rows.Count; i++)
    {
        table.Rows[i].Delete();
    }

    if (ShouldProcess(path, "RemoveItem"))
    {
        da.Update(ds, tableName);
        RemoveTable(tableName);
    }
    } //if (recurse...
    else
    {
        // Remove the table
        if (ShouldProcess(path, "RemoveItem"))
        {
            RemoveTable(tableName);
        }
    }
}
else if (type == PathType.Row)
{
    OdbcDataAdapter da = GetAdapterForTable(tableName);
    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    table.Rows[rowNumber].Delete();

    if (ShouldProcess(path, "RemoveItem"))
    {
        da.Update(ds, tableName);
    }
}
else
{
    ThrowTerminatingInvalidOperationException(path);
}

} // RemoveItem

#endregion Container Overloads

```

```

#region Helper Methods

/// <summary>
/// Checks if a given path is actually a drive name.
/// </summary>
/// <param name="path">The path to check.</param>
/// <returns>
/// True if the path given represents a drive, false otherwise.
/// </returns>
private bool PathIsDrive(string path)
{
    // Remove the drive name and first path separator. If the
    // path is reduced to nothing, it is a drive. Also if its
    // just a drive then there wont be any path separators
    if (String.IsNullOrEmpty(
        path.Replace(this.PSDriveInfo.Root, ""))
        ||
        String.IsNullOrEmpty(
            path.Replace(this.PSDriveInfo.Root + pathSeparator,
            "")))
    {
        return true;
    }
    else
    {
        return false;
    }
} // PathIsDrive

/// <summary>
/// Breaks up the path into individual elements.
/// </summary>
/// <param name="path">The path to split.</param>
/// <returns>An array of path segments.</returns>
private string[] ChunkPath(string path)
{
    // Normalize the path before splitting
    string normalPath = NormalizePath(path);

    // Return the path with the drive name and first path
    // separator character removed, split by the path separator.
    string pathNoDrive = normalPath.Replace(this.PSDriveInfo.Root
        + pathSeparator, "");

    return pathNoDrive.Split(pathSeparator.ToCharArray());
} // ChunkPath

/// <summary>
/// Adapts the path, making sure the correct path separator
/// character is used.
/// </summary>
/// <param name="path"></param>
/// <returns></returns>
private string NormalizePath(string path)

```

```

{
    string result = path;

    if (!String.IsNullOrEmpty(path))
    {
        result = path.Replace("/", pathSeparator);
    }

    return result;
} // NormalizePath

/// <summary>
/// Chunks the path and returns the table name and the row number
/// from the path
/// </summary>
/// <param name="path">Path to chunk and obtain information</param>
/// <param name="tableName">Name of the table as represented in the
/// path</param>
/// <param name="rowNumber">Row number obtained from the path</param>
/// <returns>what the path represents</returns>
private PathType GetNamesFromPath(string path, out string tableName,
out int rowNumber)
{
    PathType retVal = PathType.Invalid;
    rowNumber = -1;
    tableName = null;

    // Check if the path specified is a drive
    if (PathIsDrive(path))
    {
        return PathType.Database;
    }

    // chunk the path into parts
    string[] pathChunks = ChunkPath(path);

    switch (pathChunks.Length)
    {
        case 1:
        {
            string name = pathChunks[0];

            if (TableNameIsValid(name))
            {
                tableName = name;
                retVal = PathType.Table;
            }
        }
        break;

        case 2:
        {
            string name = pathChunks[0];

            if (TableNameIsValid(name))

```

```

        {
            tableName = name;
        }

        int number = SafeConvertRowNumber(pathChunks[1]);

        if (number >= 0)
        {
            rowNumber = number;
            retVal = PathType.Row;
        }
        else
        {
            WriteError(new ErrorRecord(
                new ArgumentException("Row number is not
valid"),
                "RowNumberNotValid",
                ErrorCategory.InvalidArgument,
                path));
        }
    }
    break;

    default:
    {
        WriteError(new ErrorRecord(
            new ArgumentException("The path supplied has too
many segments"),
            "PathNotValid",
            ErrorCategory.InvalidArgument,
            path));
    }
    break;
} // switch(pathChunks...

return retVal;
} // GetNamesFromPath

/// <summary>
/// Throws an argument exception stating that the specified path does
/// not represent either a table or a row
/// </summary>
/// <param name="path">path which is invalid</param>
private void ThrowTerminatingInvalidPathException(string path)
{
    StringBuilder message = new StringBuilder("Path must represent
either a table or a row :");
    message.Append(path);

    throw new ArgumentException(message.ToString());
}

/// <summary>
/// Retrieve the list of tables from the database.
/// </summary>

```

```

    /// <returns>
    /// Collection of DatabaseTableInfo objects, each object representing
    /// information about one database table
    /// </returns>
    private Collection<DatabaseTableInfo> GetTables()
    {
        Collection<DatabaseTableInfo> results =
            new Collection<DatabaseTableInfo>();

        // using ODBC connection to the database and get the schema of
        tables
        AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

        if (di == null)
        {
            return null;
        }

        OdbcConnection connection = di.Connection;
        DataTable dt = connection.GetSchema("Tables");
        int count;

        // iterate through all rows in the schema and create
        DatabaseTableInfo
        // objects which represents a table
        foreach (DataRow dr in dt.Rows)
        {
            String tableName = dr["TABLE_NAME"] as String;
            DataColumnCollection columns = null;

            // find the number of rows in the table
            try
            {
                String cmd = "Select count(*) from \\" + tableName +
                "\\";
                OdbcCommand command = new OdbcCommand(cmd, connection);

                count = (Int32)command.ExecuteScalar();
            }
            catch
            {
                count = 0;
            }

            // create DatabaseTableInfo object representing the table
            DatabaseTableInfo table =
                new DatabaseTableInfo(dr, tableName, count, columns);

            results.Add(table);
        } // foreach (DataRow...

        return results;
    } // GetTables

    /// <summary>

```

```

///> Return row information from a specified table.
///</summary>
///<param name="tableName">The name of the database table from
///which to retrieve rows.</param>
///<returns>Collection of row information objects.</returns>
private Collection<DatabaseRowInfo> GetRows(string tableName)
{
    Collection<DatabaseRowInfo> results =
        new Collection<DatabaseRowInfo>();

    // Obtain rows in the table and add it to the collection
    try
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);

        if (da == null)
        {
            return null;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        int i = 0;
        foreach (DataRow row in table.Rows)
        {
            results.Add(new DatabaseRowInfo(row,
i.ToString(CultureInfo.CurrentCulture)));
            i++;
        } // foreach (DataRow...
    }
    catch (Exception e)
    {
        WriteError(new ErrorRecord(e, "CannotAccessSpecifiedRows",
            ErrorCategory.InvalidOperation, tableName));
    }

    return results;
} // GetRows

///<summary>
///<Retrieve information about a single table.
///</summary>
///<param name="tableName">The table for which to retrieve
///data.</param>
///<returns>Table information.</returns>
private DatabaseTableInfo GetTable(string tableName)
{
    foreach (DatabaseTableInfo table in GetTables())
    {
        if (String.Equals(tableName, table.Name,
StringComparison.OrdinalIgnoreCase))
        {
            return table;
        }
    }
}

```

```

        }

        return null;
    } // GetTable

    /// <summary>
    /// Removes the specified table from the database
    /// </summary>
    /// <param name="tableName">Name of the table to remove</param>
    private void RemoveTable(string tableName)
    {
        // validate if tablename is valid and if table is present
        if (String.IsNullOrEmpty(tableName) ||
!TableNameIsValid(tableName) || !TableIsPresent(tableName))
        {
            return;
        }

        // Execute command using ODBC connection to remove a table
        try
        {
            // delete the table using an sql statement
            string sql = "drop table " + tableName;

            AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

            if (di == null)
            {
                return;
            }
            OdbcConnection connection = di.Connection;

            OdbcCommand cmd = new OdbcCommand(sql, connection);
            cmd.ExecuteScalar();
        }
        catch (Exception ex)
        {
            WriteError(new ErrorRecord(ex, "CannotRemoveSpecifiedTable",
ErrorCategory.InvalidOperation, null)
            );
        }
    }

} // RemoveTable

    /// <summary>
    /// Obtain a data adapter for the specified Table
    /// </summary>
    /// <param name="tableName">Name of the table to obtain the
    /// adapter for</param>
    /// <returns>Adapter object for the specified table</returns>
    /// <remarks>An adapter serves as a bridge between a DataSet (in
memory
        /// representation of table) and the data source</remarks>

```

```

private OdbcDataAdapter GetAdapterForTable(string tableName)
{
    OdbcDataAdapter da = null;
    AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

    if (di == null || !TableNameIsValid(tableName)
    || !TableIsPresent(tableName))
    {
        return null;
    }

    OdbcConnection connection = di.Connection;

    try
    {
        // Create a odbc data adpater. This can be sued to update the
        // data source with the records that will be created here
        // using data sets
        string sql = "Select * from " + tableName;
        da = new OdbcDataAdapter(new OdbcCommand(sql, connection));

        // Create a odbc command builder object. This will create sql
        // commands automatically for a single table, thus
        // eliminating the need to create new sql statements for
        // every operation to be done.
        OdbcCommandBuilder cmd = new OdbcCommandBuilder(da);

        // Set the delete cmd for the table here
        sql = "Delete from " + tableName + " where ID = ?";
        da.DeleteCommand = new OdbcCommand(sql, connection);

        // Specify a DeleteCommand parameter based on the "ID"
        // column
        da.DeleteCommand.Parameters.Add(new OdbcParameter());
        da.DeleteCommand.Parameters[0].SourceColumn = "ID";

        // Create an InsertCommand based on the sql string
        // Insert into "tablename" values (?,?,?) where
        // ? represents a column in the table. Note that
        // the number of ? will be equal to the number of
        // columnds
        DataSet ds = new DataSet();

        da.FillSchema(ds, SchemaType.Source);
        ds.Locale = CultureInfo.InvariantCulture;

        sql = "Insert into " + tableName + " values ( ";
        for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
        {
            sql += "?, ";
        }
        sql = sql.Substring(0, sql.Length - 2);
        sql += ")";
        da.InsertCommand = new OdbcCommand(sql, connection);
    }
}

```

```

        // Create parameters for the InsertCommand based on the
        // captions of each column
        for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
        {
            da.InsertCommand.Parameters.Add(new OdbcParameter());
            da.InsertCommand.Parameters[i].SourceColumn =
                ds.Tables["Table"].Columns[i].Caption;

        }

        // Open the connection if its not already open
        if (connection.State != ConnectionState.Open)
        {
            connection.Open();
        }
    }
    catch (Exception e)
    {
        WriteError(new ErrorRecord(e, "CannotAccessSpecifiedTable",
            ErrorCategory.InvalidOperation, tableName));
    }

    return da;
} // GetAdapterForTable

/// <summary>
/// Gets the DataSet (in memory representation) for the table
/// for the specified adapter
/// </summary>
/// <param name="adapter">Adapter to be used for obtaining
/// the table</param>
/// <param name="tableName">Name of the table for which a
/// DataSet is required</param>
/// <returns>The DataSet with the filled in schema</returns>
private DataSet GetDataSetForTable(OdbcDataAdapter adapter, string
tableName)
{
    Debug.Assert(adapter != null);

    // Create a dataset object which will provide an in-memory
    // representation of the data being worked upon in the
    // data source.
    DataSet ds = new DataSet();

    // Create a table named "Table" which will contain the same
    // schema as in the data source.
    //adapter.FillSchema(ds, SchemaType.Source);
    adapter.Fill(ds, tableName);
    ds.Locale = CultureInfo.InvariantCulture;

    return ds;
} //GetDataSetForTable

/// <summary>
/// Get the DataTable object which can be used to operate on

```

```

/// for the specified table in the data source
/// </summary>
/// <param name="ds">DataSet object which contains the tables
/// schema</param>
/// <param name="tableName">Name of the table</param>
/// <returns>Corresponding DataTable object representing
/// the table</returns>
///
private DataTable GetDataTable(DataSet ds, string tableName)
{
    Debug.Assert(ds != null);
    Debug.Assert(tableName != null);

    DataTable table = ds.Tables[tableName];
    table.Locale = CultureInfo.InvariantCulture;

    return table;
} // GetDataTable

/// <summary>
/// Retrieves a single row from the named table.
/// </summary>
/// <param name="tableName">The table that contains the
/// numbered row.</param>
/// <param name="row">The index of the row to return.</param>
/// <returns>The specified table row.</returns>
private DatabaseRowInfo GetRow(string tableName, int row)
{
    Collection<DatabaseRowInfo> di = GetRows(tableName);

    // if the row is invalid write an appropriate error else return
the
    // corresponding row information
    if (row < di.Count && row >= 0)
    {
        return di[row];
    }
    else
    {
        WriteError(new ErrorRecord(
            new ItemNotFoundException(),
            "RowNotFound",
            ErrorCategory.ObjectNotFound,
            row.ToString(CultureInfo.CurrentCulture))
        );
    }

    return null;
} // GetRow

/// <summary>
/// Method to safely convert a string representation of a row number
/// into its Int32 equivalent
/// </summary>
/// <param name="rowNumberAsStr">String representation of the row

```

```

    ///> number</param>
    ///> If there is an exception, -1 is returned</remarks>
    private int SafeConvertRowNumber(string rowNumberAsStr)
    {
        int rowNumber = -1;
        try
        {
            rowNumber = Convert.ToInt32(rowNumberAsStr,
CultureInfo.CurrentCulture);
        }
        catch (FormatException fe)
        {
            WriteError(new ErrorRecord(fe, "RowStringFormatNotValid",
                ErrorCategory.InvalidData, rowNumberAsStr));
        }
        catch (OverflowException oe)
        {
            WriteError(new ErrorRecord(oe,
"RowStringConversionToNumberFailed",
                ErrorCategory.InvalidData, rowNumberAsStr));
        }

        return rowNumber;
    } // SafeConvertRowNumber

    ///> summary>
    ///> Check if a table name is valid
    ///> </summary>
    ///> <param name="tableName">Table name to validate</param>
    ///> <remarks>Helps to check for SQL injection attacks</remarks>
    private bool TableNameIsValid(string tableName)
    {
        Regex exp = new Regex(pattern, RegexOptions.Compiled |
RegexOptions.IgnoreCase);

        if (exp.IsMatch(tableName))
        {
            return true;
        }
        WriteError(new ErrorRecord(
            new ArgumentException("Table name not valid"),
"TableNameNotValid",
                ErrorCategory.InvalidArgument, tableName));
        return false;
    } // TableNameIsValid

    ///> summary>
    ///> Checks to see if the specified table is present in the
    ///> database
    ///> </summary>
    ///> <param name="tableName">Name of the table to check</param>
    ///> <returns>true, if table is present, false otherwise</returns>
    private bool TableIsPresent(string tableName)
    {
        // using ODBC connection to the database and get the schema of

```

```

tables

    AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;
    if (di == null)
    {
        return false;
    }

    OdbcConnection connection = di.Connection;
    DataTable dt = connection.GetSchema("Tables");

    // check if the specified tableName is available
    // in the list of tables present in the database
    foreach (DataRow dr in dt.Rows)
    {
        string name = dr["TABLE_NAME"] as string;
        if (name.Equals(tableName,
StringComparison.OrdinalIgnoreCase))
        {
            return true;
        }
    }

    WriteError(new ErrorRecord(
        new ArgumentException("Specified Table is not present in
database"), "TableNotAvailable",
        ErrorCategory.InvalidArgument, tableName));

    return false;
// TableIsPresent

/// <summary>
/// Gets the next available ID in the table
/// </summary>
/// <param name="table">DataTable object representing the table to
/// search for ID</param>
/// <returns>next available id</returns>
private int GetNextID(DataTable table)
{
    int big = 0;
    int id = 0;

    for (int i = 0; i < table.Rows.Count; i++)
    {
        DataRow row = table.Rows[i];

        object o = row["ID"];

        if (o.GetType().Name.Equals("Int16"))
        {
            id = (int)(short)o;
        }
        else
        {
            id = (int)o;
        }
}

```

```

        if (big < id)
        {
            big = id;
        }
    }

    big++;
    return big;
}

#endregion Helper Methods

#region Private Properties

private string pathSeparator = "\\";
private static string pattern = @"^+[a-z]+[0-9]*_*$";

private enum PathType { Database, Table, Row, Invalid };

#endregion Private Properties
}

#endregion AccessDBProvider

#region Helper Classes

#region AccessDBPSDriveInfo

/// <summary>
/// Any state associated with the drive should be held here.
/// In this case, it's the connection to the database.
/// </summary>
internal class AccessDBPSDriveInfo : PSDriveInfo
{
    private OdbcConnection connection;

    /// <summary>
    /// ODBC connection information.
    /// </summary>
    public OdbcConnection Connection
    {
        get { return connection; }
        set { connection = value; }
    }

    /// <summary>
    /// Constructor that takes one argument
    /// </summary>
    /// <param name="driveInfo">Drive provided by this provider</param>
    public AccessDBPSDriveInfo(PSDriveInfo driveInfo)
        : base(driveInfo)
    {
    }

} // class AccessDBPSDriveInfo

```

```
#endregion AccessDBPSDriveInfo

#region DatabaseTableInfo

/// <summary>
/// Contains information specific to the database table.
/// Similar to the DirectoryInfo class.
/// </summary>
public class DatabaseTableInfo
{
    /// <summary>
    /// Row from the "tables" schema
    /// </summary>
    public DataRow Data
    {
        get
        {
            return data;
        }
        set
        {
            data = value;
        }
    }
    private DataRow data;

    /// <summary>
    /// The table name.
    /// </summary>
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
    private String name;

    /// <summary>
    /// The number of rows in the table.
    /// </summary>
    public int RowCount
    {
        get
        {
            return rowCount;
        }
        set
        {
            rowCount = value;
        }
    }
}
```

```

        }
    }

    private int rowCount;

    /// <summary>
    /// The column definitions for the table.
    /// </summary>
    public DataColumnCollection Columns
    {
        get
        {
            return columns;
        }
        set
        {
            columns = value;
        }
    }

    private DataColumnCollection columns;

    /// <summary>
    /// Constructor.
    /// </summary>
    /// <param name="row">The row definition.</param>
    /// <param name="name">The table name.</param>
    /// <param name="rowCount">The number of rows in the table.</param>
    /// <param name="columns">Information on the column tables.</param>
    public DatabaseTableInfo(DataRow row, string name, int rowCount,
                             DataColumnCollection columns)
    {
        Name = name;
        Data = row;
        RowCount = rowCount;
        Columns = columns;
    } // DatabaseTableInfo
} // class DatabaseTableInfo

#endregion DatabaseTableInfo

#region DatabaseRowInfo

    /// <summary>
    /// Contains information specific to an individual table row.
    /// Analogous to the FileInfo class.
    /// </summary>
    public class DatabaseRowInfo
    {
        /// <summary>
        /// Row data information.
        /// </summary>
        public DataRow Data
        {
            get
            {
                return data;
            }
        }
    }
}

```

```
        }
        set
        {
            data = value;
        }
    }
    private DataRow data;

    /// <summary>
    /// The row index.
    /// </summary>
    public string RowNumber
    {
        get
        {
            return RowNumber;
        }
        set
        {
            RowNumber = value;
        }
    }
    private string RowNumber;

    /// <summary>
    /// Constructor.
    /// </summary>
    /// <param name="row">The row information.</param>
    /// <param name="name">The row index.</param>
    public DatabaseRowInfo(DataRow row, string name)
    {
        RowNumber = name;
        Data = row;
    } // DatabaseRowInfo
} // class DatabaseRowInfo

#endregion DatabaseRowInfo

#endregion Helper Classes
}
```

See Also

[Windows PowerShell SDK](#)

 Collaborate with us on
GitHub



PowerShell feedback

PowerShell is an open source
project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

AccessDbProviderSample05 Code Sample

Article • 09/17/2021

The following code shows the implementation of the Windows PowerShell navigation provider described in [Creating a Windows PowerShell Navigation Provider](#). This provider supports recursive commands, nested containers, and relative paths that allow it to navigate the data store.

Code Sample

C#

```
using System;
using System.IO;
using System.Data;
using System.Data.Odbc;
using System.Diagnostics;
using System.Collections.ObjectModel;
using System.Text;
using System.Text.RegularExpressions;
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.ComponentModel;
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

        /// <summary>
        /// This example implements the navigation methods.
        /// </summary>
        [CmdletProvider("AccessDB", ProviderCapabilities.None)]
        public class AccessDBProvider : NavigationCmdletProvider
    {

        #region Drive Manipulation

            /// <summary>
            /// Create a new drive. Create a connection to the database file and
            /// set
            /// the Connection property in the PSDriveInfo.
            /// </summary>
            /// <param name="drive">
            /// Information describing the drive to add.
            /// </param>
            /// <returns>The added drive.</returns>
        
```

```

protected override PSDriveInfo NewDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            null)
        );
    }

    return null;
}

// check if drive root is not null or empty
// and if its an existing file
if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root)
== false))
{
    WriteError(new ErrorRecord(
        new ArgumentException("drive.Root"),
        "NoRoot",
        ErrorCategory.InvalidArgument,
        drive)
    );

    return null;
}

// create a new drive and create an ODBC connection to the new
drive
AccessDBPSDriveInfo accessDBPSDriveInfo = new
AccessDBPSDriveInfo(drive);

OdbcConnectionStringBuilder builder = new
OdbcConnectionStringBuilder();

builder.Driver = "Microsoft Access Driver (*.mdb)";
builder.Add("DBQ", drive.Root);

OdbcConnection conn = new
OdbcConnection(builder.ConnectionString);
conn.Open();
accessDBPSDriveInfo.Connection = conn;

return accessDBPSDriveInfo;
} // NewDrive

/// <summary>
/// Removes a drive from the provider.
/// </summary>
/// <param name="drive">The drive to remove.</param>
/// <returns>The drive removed.</returns>

```

```

protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            drive)
        );
    }

    return null;
}

// close ODBC connection to the drive
AccessDBPSDriveInfo accessDBPSDriveInfo = drive as
AccessDBPSDriveInfo;

if (accessDBPSDriveInfo == null)
{
    return null;
}
accessDBPSDriveInfo.Connection.Close();

return accessDBPSDriveInfo;
} // RemoveDrive

#endregion Drive Manipulation

#region Item Methods

/// <summary>
/// Retrieves an item using the specified path.
/// </summary>
/// <param name="path">The path to the item to return.</param>
protected override void GetItem(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        WriteItemObject(this.PSDriveInfo, path, true);
        return;
    } // if (PathIsDrive...

    // Get table name and row information from the path and do
    // necessary actions
    string tableName;
    int lineNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
    lineNumber);

    if (type == PathType.Table)
    {

```

```

        DatabaseTableInfo table = GetTable(tableName);
        WriteItemObject(table, path, true);
    }
    else if (type == PathType.Row)
    {
        DatabaseRowInfo row = GetRow(tableName, rowNumber);
        WriteItemObject(row, path, false);
    }
    else
    {
        ThrowTerminatingInvalidPathException(path);
    }

} // GetItem

/// <summary>
/// Set the content of a row of data specified by the supplied path
/// parameter.
/// </summary>
/// <param name="path">Specifies the path to the row whose columns
/// will be updated.</param>
/// <param name="values">Comma separated string of values</param>
protected override void SetItem(string path, object values)
{
    // Get type, table name and row number from the path specified
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type != PathType.Row)
    {
        WriteError(new ErrorRecord(new NotSupportedException(
            "SetNotSupported"), "", 
        ErrorCategory.InvalidOperation, path));

        return;
    }

    // Get in-memory representation of table
    OdbcDataAdapter da = GetAdapterForTable(tableName);

    if (da == null)
    {
        return;
    }
    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    if (rowNumber >= table.Rows.Count)
    {
        // The specified row number has to be available. If not
        // NewItem has to be used to add a new row
        throw new ArgumentException("Row specified is not

```

```

available");
} // if (rowNum...

string[] colValues = (values as string).Split(',');
// set the specified row
DataRow row = table.Rows[rowNumber];

for (int i = 0; i < colValues.Length; i++)
{
    row[i] = colValues[i];
}

// Update the table
if (ShouldProcess(path, "SetItem"))
{
    da.Update(ds, tableName);
}

} // SetItem

/// <summary>
/// Test to see if the specified item exists.
/// </summary>
/// <param name="path">The path to the item to verify.</param>
/// <returns>True if the item is found.</returns>
protected override bool ItemExists(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        return true;
    }

    // Obtain type, table name and row number from path
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    DatabaseTableInfo table = GetTable(tableName);

    if (type == PathType.Table)
    {
        // if specified path represents a table then
DatabaseTableInfo
        // object for the same should exist
        if (table != null)
        {
            return true;
        }
    }
    else if (type == PathType.Row)
    {
}

```

```

        // if specified path represents a row then DatabaseTableInfo
should
        // exist for the table and then specified row number must be
within
        // the maximum row count in the table
        if (table != null && rowNum < table.RowCount)
        {
            return true;
        }
    }

    return false;

} // ItemExists

/// <summary>
/// Test to see if the specified path is syntactically valid.
/// </summary>
/// <param name="path">The path to validate.</param>
/// <returns>True if the specified path is valid.</returns>
protected override bool IsValidPath(string path)
{
    bool result = true;

    // check if the path is null or empty
    if (String.IsNullOrEmpty(path))
    {
        result = false;
    }

    // convert all separators in the path to a uniform one
    path = NormalizePath(path);

    // split the path into individual chunks
    string[] pathChunks = path.Split(pathSeparator.ToCharArray());

    foreach (string pathChunk in pathChunks)
    {
        if (pathChunk.Length == 0)
        {
            result = false;
        }
    }
    return result;
} // IsValidPath

#endregion Item Overloads

#region Container Overloads

/// <summary>
/// Return either the tables in the database or the datarows
/// </summary>
/// <param name="path">The path to the parent</param>
/// <param name="recurse">True to return all child items recursively.

```

```

/// </param>
protected override void GetChildItems(string path, bool recurse)
{
    // If path represented is a drive then the children in the path
are
    // tables. Hence all tables in the drive represented will have to
be
    // returned
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table, path, true);

            // if the specified item exists and recurse has been set
then
            // all child items within it have to be obtained as well
            if (ItemExists(path) && recurse)
            {
                GetChildItems(path + pathSeparator + table.Name,
recurse);
            }
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
else
{
    // Get the table name, row number and type of path from the
    // path specified
    string tableName;
    int rowCount;

    PathType type = GetNamesFromPath(path, out tableName, out
rowCount);

    if (type == PathType.Table)
    {
        // Obtain all the rows within the table
        foreach (DatabaseRowInfo row in GetRows(tableName))
        {
            WriteItemObject(row, path + pathSeparator +
row.RowNumber,
                            false);
        } // foreach (DatabaseRowInfo...
    }
    else if (type == PathType.Row)
    {
        // In this case the user has directly specified a row,
hence
        // just give that particular row
        DatabaseRowInfo row = GetRow(tableName, rowCount);
        WriteItemObject(row, path + pathSeparator +
row.RowNumber,
                            false);
    }
    else

```

```

    {
        // In this case, the path specified is not valid
        ThrowTerminatingInvalidOperationException(path);
    }
} // else
} // GetChildItems

/// <summary>
/// Return the names of all child items.
/// </summary>
/// <param name="path">The root path.</param>
/// <param name="returnContainers">Not used.</param>
protected override void GetChildNames(string path,
                                      ReturnContainers returnContainers)
{
    // If the path represented is a drive, then the child items are
    // tables. get the names of all the tables in the drive.
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table.Name, path, true);
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get type, table name and row number from path specified
        string tableName;
        int rowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Table)
        {
            // Get all the rows in the table and then write out the
            // row numbers.
            foreach (DatabaseRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row.RowNumber, path, false);
            } // foreach (DatabaseRowInfo...
        }
        else if (type == PathType.Row)
        {
            // In this case the user has directly specified a row,
            hence
            // just give that particular row
            DatabaseRowInfo row = GetRow(tableName, rowNumber);

            WriteItemObject(row.RowNumber, path, false);
        }
        else
        {
            ThrowTerminatingInvalidOperationException(path);
        }
    }
}

```

```

        } // else
    } // GetChildNames

    /// <summary>
    /// Determines if the specified path has child items.
    /// </summary>
    /// <param name="path">The path to examine.</param>
    /// <returns>
    /// True if the specified path has child items.
    /// </returns>
    protected override bool HasChildItems(string path)
    {
        if (PathIsDrive(path))
        {
            return true;
        }

        return (ChunkPath(path).Length == 1);
    } // HasChildItems

    /// <summary>
    /// Creates a new item at the specified path.
    /// </summary>
    ///
    /// <param name="path">
    /// The path to the new item.
    /// </param>
    ///
    /// <param name="type">
    /// Type for the object to create. "Table" for creating a new table
and
    /// "Row" for creating a new row in a table.
    /// </param>
    ///
    /// <param name="newValue">
    /// Object for creating new instance of a type at the specified path.
For
    /// creating a "Table" the object parameter is ignored and for
creating
    /// a "Row" the object must be of type string which will contain
comma
    /// separated values of the rows to insert.
    /// </param>
    protected override void NewItem(string path, string type,
                                    object newValue)
    {
        string tableName;
        int rowNumber;

        PathType pt = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (pt == PathType.Invalid)
        {
            ThrowTerminatingInvalidPathException(path);
        }
    }
}

```

```

        }

        // Check if type is either "table" or "row", if not throw an
        // exception
        if (!String.Equals(type, "table",
 StringComparison.OrdinalIgnoreCase)
            && !String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
        {
            WriteError(new ErrorRecord
                (new ArgumentException("Type must be either
a table or row"),
                 "CannotCreateSpecifiedObject",
                 ErrorCategory.InvalidArgument,
                 path
                )
            );
        }

        throw new ArgumentException("This provider can only create
items of type \"table\" or \"row\"");
    }

    // Path type is the type of path of the container. So if a drive
    // is specified, then a table can be created under it and if a
    table
    of
    by
    row
    based
    // is specified, then a row can be created under it. For the sake
    // completeness, if a row is specified, then if the row specified
    // the path does not exist, a new row is created. However, the
    // number may not match as the row numbers only get incremented
    // on the number of rows

    if (PathIsDrive(path))
    {
        if (String.Equals(type, "table",
 StringComparison.OrdinalIgnoreCase))
        {
            // Execute command using ODBC connection to create a
            table
            try
            {
                // create the table using an sql statement
                string newTableName = newItemValue.ToString();
                string sql = "create table " + newTableName
                    + " (ID INT)";

                // Create the table using the Odbc connection from
                the
                // drive.
                AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;
            }
        }
    }
}

```

```

        if (di == null)
        {
            return;
        }
        OdbcConnection connection = di.Connection;

        if (ShouldProcess(newTableName, "create"))
        {
            OdbcCommand cmd = new OdbcCommand(sql,
connection);
            cmd.ExecuteScalar();
        }
    }
    catch (Exception ex)
    {
        WriteError(new ErrorRecord(ex,
"CannotCreateSpecifiedTable",
ErrorCategory.InvalidOperation, path)
);
    }
} // if (String...
else if (String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
{
    throw new
        ArgumentException("A row cannot be created under a
database, specify a path that represents a Table");
}
}// if (PathIsDrive...
else
{
    if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
    {
        if (rowNumber < 0)
        {
            throw new
                ArgumentException("A table cannot be created
within another table, specify a path that represents a database");
        }
        else
        {
            throw new
                ArgumentException("A table cannot be created
inside a row, specify a path that represents a database");
        }
    } //if (String.Equals....
// if path specified is a row, create a new row
else if (String.Equals(type, "row",
StringComparison.OrdinalIgnoreCase))
{
    // The user is required to specify the values to be
inserted
    // into the table in a single string separated by commas
    string value = newItemValue as string;
}
}

```

```

        if (String.IsNullOrEmpty(value))
        {
            throw new
                ArgumentException("Value argument must have comma
separated values of each column in a row");
        }
        string[] rowValues = value.Split(',');

        OdbcDataAdapter da = GetAdapterForTable(tableName);

        if (da == null)
        {
            return;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        if (rowValues.Length != table.Columns.Count)
        {
            string message =
String.Format(CultureInfo.CurrentCulture,
                           "The table has {0} columns and
the value specified must have so many comma separated values",
                           table.Columns.Count);

            throw new ArgumentException(message);
        }

        if (!Force && (rowNumber >= 0 && rowNumber <
table.Rows.Count))
        {
            string message =
String.Format(CultureInfo.CurrentCulture,
                           "The row {0} already exists. To
create a new row specify row number as {1}, or specify path to a table, or
use the -Force parameter",
                           rowNumber,
                           table.Rows.Count);

            throw new ArgumentException(message);
        }

        if (rowNumber > table.Rows.Count)
        {
            string message =
String.Format(CultureInfo.CurrentCulture,
                           "To create a new row specify row
number as {0}, or specify path to a table",
                           table.Rows.Count);

            throw new ArgumentException(message);
        }
    }
}

```

```

        // Create a new row and update the row with the input
        // provided by the user
        DataRow row = table.NewRow();
        for (int i = 0; i < rowValues.Length; i++)
        {
            row[i] = rowValues[i];
        }
        table.Rows.Add(row);

        if (ShouldProcess(tableName, "update rows"))
        {
            // Update the table from memory back to the data
source
            da.Update(ds, tableName);
        }

        }// else if (String...
}// else ...

} // NewItem

/// <summary>
/// Copies an item at the specified path to the location specified
/// </summary>
///
/// <param name="path">
/// Path of the item to copy
/// </param>
///
/// <param name="copyPath">
/// Path of the item to copy to
/// </param>
///
/// <param name="recurse">
/// Tells the provider to recurse subcontainers when copying
/// </param>
///
protected override void CopyItem(string path, string copyPath, bool
recurse)
{
    string tableName, copyTableName;
    int rowNumber, copyRowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);
    PathType copyType = GetNamesFromPath(copyPath, out copyTableName,
out copyRowNumber);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }

    if (type == PathType.Invalid)
    {

```

```

        ThrowTerminatingInvalidOperationException(copyPath);
    }

    // Get the table and the table to copy to
    OdbcDataAdapter da = GetAdapterForTable(tableName);
    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    OdbcDataAdapter cda = GetAdapterForTable(copyTableName);
    if (cda == null)
    {
        return;
    }

    DataSet cds = GetDataSetForTable(cda, copyTableName);
    DataTable copyTable = GetDataTable(cds, copyTableName);

    // if source represents a table
    if (type == PathType.Table)
    {
        // if copyPath does not represent a table
        if (copyType != PathType.Table)
        {
            ArgumentException e = new ArgumentException("Table can
only be copied on to another table location");

            WriteError(new ErrorRecord(e, "PathNotValid",
                ErrorCategory.InvalidArgument, copyPath));

            throw e;
        }

        // if table already exists then force parameter should be set
        // to force a copy
        if (!Force && GetTable(copyTableName) != null)
        {
            throw new ArgumentException("Specified path already
exists");
        }

        for (int i = 0; i < table.Rows.Count; i++)
        {
            DataRow row = table.Rows[i];
            DataRow copyRow = copyTable.NewRow();

            copyRow.ItemArray = row.ItemArray;
            copyTable.Rows.Add(copyRow);
        }
    } // if (type == ...
    // if source represents a row

```

```

        else
        {
            if (copyType == PathType.Row)
            {
                if (!Force && (copyRowNumber < copyTable.Rows.Count))
                {
                    throw new ArgumentException("Specified path already
exists.");
                }

                DataRow row = table.Rows[rowNumber];
                DataRow copyRow = null;

                if (copyRowNumber < copyTable.Rows.Count)
                {
                    // copy to an existing row
                    copyRow = copyTable.Rows[copyRowNumber];
                    copyRow.ItemArray = row.ItemArray;
                    copyRow[0] = GetNextID(copyTable);
                }
                else if (copyRowNumber == copyTable.Rows.Count)
                {
                    // copy to the next row in the table that will
                    // be created
                    copyRow = copyTable.NewRow();
                    copyRow.ItemArray = row.ItemArray;
                    copyRow[0] = GetNextID(copyTable);
                    copyTable.Rows.Add(copyRow);
                }
                else
                {
                    // attempting to copy to a nonexistent row or a row
                    // that cannot be created now - throw an exception
                    string message =
String.Format(CultureInfo.CurrentCulture,
                           "The item cannot be specified to
the copied row. Specify row number as {0}, or specify a path to the table.",

                           table.Rows.Count);

                    throw new ArgumentException(message);
                }
            }
            else
            {
                // destination path specified represents a table,
                // create a new row and copy the item
                DataRow copyRow = copyTable.NewRow();
                copyRow.ItemArray = table.Rows[rowNumber].ItemArray;
                copyRow[0] = GetNextID(copyTable);
                copyTable.Rows.Add(copyRow);
            }
        }

        if (ShouldProcess(copyTableName, "CopyItems"))
        {

```

```

        cda.Update(cds, copyTableName);
    }

} //CopyItem

/// <summary>
/// Removes (deletes) the item at the specified path
/// </summary>
///
/// <param name="path">
/// The path to the item to remove.
/// </param>
///
/// <param name="recurse">
/// True if all children in a subtree should be removed, false if
only
    /// the item at the specified path should be removed. Is applicable
    /// only for container (table) items. Its ignored otherwise (even if
    /// specified).
    /// </param>
    ///
    /// <remarks>
    /// There are no elements in this store which are hidden from the
user.
    /// Hence this method will not check for the presence of the Force
    /// parameter
    /// </remarks>
    ///
protected override void RemoveItem(string path, bool recurse)
{
    string tableName;
    int rowNumber = 0;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        // if recurse flag has been specified, delete all the rows as
well
        if (recurse)
        {
            OdbcDataAdapter da = GetAdapterForTable(tableName);
            if (da == null)
            {
                return;
            }

            DataSet ds = GetDataSetForTable(da, tableName);
            DataTable table = GetDataTable(ds, tableName);

            for (int i = 0; i < table.Rows.Count; i++)
            {
                table.Rows[i].Delete();
            }
        }
    }
}

```

```

        if (ShouldProcess(path, "RemoveItem"))
        {
            da.Update(ds, tableName);
            RemoveTable(tableName);
        }
    } //if (recurse...
    else
    {
        // Remove the table
        if (ShouldProcess(path, "RemoveItem"))
        {
            RemoveTable(tableName);
        }
    }
}
else if (type == PathType.Row)
{
    OdbcDataAdapter da = GetAdapterForTable(tableName);
    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    table.Rows[rowNumber].Delete();

    if (ShouldProcess(path, "RemoveItem"))
    {
        da.Update(ds, tableName);
    }
}
else
{
    ThrowTerminatingInvalidOperationException(path);
}

} // RemoveItem

#endregion Container Overloads

#region Navigation

/// <summary>
/// Determine if the path specified is that of a container.
/// </summary>
/// <param name="path">The path to check.</param>
/// <returns>True if the path specifies a container.</returns>
protected override bool IsItemContainer(string path)
{
    if (PathIsDrive(path))
    {
        return true;
    }
}

```

```

        }

        string[] pathChunks = ChunkPath(path);
        string tableName;
        int rowCount;

        PathType type = GetNamesFromPath(path, out tableName, out
rowCount);

        if (type == PathType.Table)
        {
            foreach (DatabaseTableInfo ti in GetTables())
            {
                if (string.Equals(ti.Name, tableName,
StringComparison.OrdinalIgnoreCase))
                {
                    return true;
                }
            } // foreach (DatabaseTableInfo...
        } // if (pathChunks...

        return false;
    } // IsItemContainer

    /// <summary>
    /// Get the name of the leaf element in the specified path
    /// </summary>
    ///
    /// <param name="path">
    /// The full or partial provider specific path
    /// </param>
    ///
    /// <returns>
    /// The leaf element in the path
    /// </returns>
    protected override string GetChildName(string path)
    {
        if (PathIsDrive(path))
        {
            return path;
        }

        string tableName;
        int rowCount;

        PathType type = GetNamesFromPath(path, out tableName, out
rowCount);

        if (type == PathType.Table)
        {
            return tableName;
        }
        else if (type == PathType.Row)
        {
            return rowCount.ToString(CultureInfo.CurrentCulture);
        }
    }
}

```

```
        }
        else
        {
            ThrowTerminatingInvalidPathException(path);
        }

        return null;
    }

    /// <summary>
    /// Removes the child segment of the path and returns the remaining
    /// parent portion
    /// </summary>
    ///
    /// <param name="path">
    /// A full or partial provider specific path. The path may be to an
    /// item that may or may not exist.
    /// </param>
    ///
    /// <param name="root">
    /// The fully qualified path to the root of a drive. This parameter
    /// may be null or empty if a mounted drive is not in use for this
    /// operation. If this parameter is not null or empty the result
    /// of the method should not be a path to a container that is a
    /// parent or in a different tree than the root.
    /// </param>
    ///
    /// <returns></returns>

protected override string GetParentPath(string path, string root)
{
    // If root is specified then the path has to contain
    // the root. If not nothing should be returned
    if (!String.IsNullOrEmpty(root))
    {
        if (!path.Contains(root))
        {
            return null;
        }
    }

    return path.Substring(0, path.LastIndexOf(pathSeparator,
 StringComparison.OrdinalIgnoreCase));
}

    /// <summary>
    /// Joins two strings with a provider specific path separator.
    /// </summary>
    ///
    /// <param name="parent">
    /// The parent segment of a path to be joined with the child.
    /// </param>
    ///
    /// <param name="child">
    /// The child segment of a path to be joined with the parent.
}
```

```
    /// </param>
    ///
    /// <returns>
    /// A string that represents the parent and child segments of the
    path
    /// joined by a path separator.
    /// </returns>

    protected override string MakePath(string parent, string child)
    {
        string result;

        string normalParent = NormalizePath(parent);
        normalParent = RemoveDriveFromPath(normalParent);
        string normalChild = NormalizePath(child);
        normalChild = RemoveDriveFromPath(normalChild);

        if (String.IsNullOrEmpty(normalParent) &&
String.IsNullOrEmpty(normalChild))
        {
            result = String.Empty;
        }
        else if (String.IsNullOrEmpty(normalParent) &&
!String.IsNullOrEmpty(normalChild))
        {
            result = normalChild;
        }
        else if (!String.IsNullOrEmpty(normalParent) &&
String.IsNullOrEmpty(normalChild))
        {
            if (normalParent.EndsWith(pathSeparator,
 StringComparison.OrdinalIgnoreCase))
            {
                result = normalParent;
            }
            else
            {
                result = normalParent + pathSeparator;
            }
        } // else if (!String...
        else
        {
            if (!normalParent.Equals(String.Empty) &&
                !normalParent.EndsWith(pathSeparator,
StringComparison.OrdinalIgnoreCase))
            {
                result = normalParent + pathSeparator;
            }
            else
            {
                result = normalParent;
            }

            if (normalChild.StartsWith(pathSeparator,
StringComparison.OrdinalIgnoreCase))
```

```

        {
            result += normalChild.Substring(1);
        }
        else
        {
            result += normalChild;
        }
    } // else

    return result;
} // MakePath

/// <summary>
/// Normalizes the path that was passed in and returns the normalized
/// path as a relative path to the basePath that was passed.
/// </summary>
///
/// <param name="path">
/// A fully qualified provider specific path to an item. The item
/// should exist or the provider should write out an error.
/// </param>
///
/// <param name="basepath">
/// The path that the return value should be relative to.
/// </param>
///
/// <returns>
/// A normalized path that is relative to the basePath that was
/// passed. The provider should parse the path parameter, normalize
/// the path, and then return the normalized path relative to the
/// basePath.
/// </returns>

protected override string NormalizeRelativePath(string path,
                                                string basepath)
{
    // Normalize the paths first
    string normalPath = NormalizePath(path);
    normalPath = RemoveDriveFromPath(normalPath);
    string normalBasePath = NormalizePath(basepath);
    normalBasePath = RemoveDriveFromPath(normalBasePath);

    if (String.IsNullOrEmpty(normalBasePath))
    {
        return normalPath;
    }
    else
    {
        if (!normalPath.Contains(normalBasePath))
        {
            return null;
        }

        return normalPath.Substring(normalBasePath.Length +
pathSeparator.Length);
    }
}

```

```
        }

    }

    /// <summary>
    /// Moves the item specified by the path to the specified destination
    /// </summary>
    ///
    /// <param name="path">
    /// The path to the item to be moved
    /// </param>
    ///
    /// <param name="destination">
    /// The path of the destination container
    /// </param>

    protected override void MoveItem(string path, string destination)
    {
        // Get type, table name and rowNumber from the path
        string tableName, destTableName;
        int rowNumber, destRowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        PathType destType = GetNamesFromPath(destination, out
destTableName,
                                         out destRowNumber);

        if (type == PathType.Invalid)
        {
            ThrowTerminatingInvalidPathException(path);
        }

        if (destType == PathType.Invalid)
        {
            ThrowTerminatingInvalidPathException(destination);
        }

        if (type == PathType.Table)
        {
            ArgumentException e = new ArgumentException("Move not
supported for tables");

            WriteError(new ErrorRecord(e, "MoveNotSupported",
ErrorCategory.InvalidArgument, path));

            throw e;
        }
        else
        {
            OdbcDataAdapter da = GetAdapterForTable(tableName);
            if (da == null)
            {
                return;
            }
        }
    }
}
```

```

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        OdbcDataAdapter dda = GetAdapterForTable(destTableName);
        if (dda == null)
        {
            return;
        }

        DataSet dds = GetDataSetForTable(dda, destTableName);
        DataTable destTable = GetDataTable(dds, destTableName);
        DataRow row = table.Rows[rowNumber];

        if (destType == PathType.Table)
        {
            DataRow destRow = destTable.NewRow();

            destRow.ItemArray = row.ItemArray;
        }
        else
        {
            DataRow destRow = destTable.Rows[destRowNumber];

            destRow.ItemArray = row.ItemArray;
        }

        // Update the changes
        if (ShouldProcess(path, "MoveItem"))
        {
            WriteItemObject(row, path, false);
            dda.Update(dds, destTableName);
        }
    }
}

#endregion Navigation

#region Helper Methods

/// <summary>
/// Checks if a given path is actually a drive name.
/// </summary>
/// <param name="path">The path to check.</param>
/// <returns>
/// True if the path given represents a drive, false otherwise.
/// </returns>
private bool PathIsDrive(string path)
{
    // Remove the drive name and first path separator. If the
    // path is reduced to nothing, it is a drive. Also if its
    // just a drive then there wont be any path separators
    if (String.IsNullOrEmpty(
        path.Replace(this.PSDriveInfo.Root, "")) ||
        String.IsNullOrEmpty(

```

```

                path.Replace(this.PSDriveInfo.Root + pathSeparator,
                ""))
        {
            return true;
        }
        else
        {
            return false;
        }
    } // PathIsDrive

    /// <summary>
    /// Breaks up the path into individual elements.
    /// </summary>
    /// <param name="path">The path to split.</param>
    /// <returns>An array of path segments.</returns>
    private string[] ChunkPath(string path)
    {
        // Normalize the path before splitting
        string normalPath = NormalizePath(path);

        // Return the path with the drive name and first path
        // separator character removed, split by the path separator.
        string pathNoDrive = normalPath.Replace(this.PSDriveInfo.Root
            + pathSeparator, "");

        return pathNoDrive.Split(pathSeparator.ToCharArray());
    } // ChunkPath

    /// <summary>
    /// Adapts the path, making sure the correct path separator
    /// character is used.
    /// </summary>
    /// <param name="path"></param>
    /// <returns></returns>
    private string NormalizePath(string path)
    {
        string result = path;

        if (!String.IsNullOrEmpty(path))
        {
            result = path.Replace("/", pathSeparator);
        }

        return result;
    } // NormalizePath

    /// <summary>
    /// Ensures that the drive is removed from the specified path
    /// </summary>
    ///
    /// <param name="path">Path from which drive needs to be
    removed</param>

```

```

/// <returns>Path with drive information removed</returns>
private string RemoveDriveFromPath(string path)
{
    string result = path;
    string root;

    if (this.PSDriveInfo == null)
    {
        root = String.Empty;
    }
    else
    {
        root = this.PSDriveInfo.Root;
    }

    if (result == null)
    {
        result = String.Empty;
    }

    if (result.Contains(root))
    {
        result = result.Substring(result.IndexOf(root,
StringComparison.OrdinalIgnoreCase) + root.Length);
    }

    return result;
}

/// <summary>
/// Chunks the path and returns the table name and the row number
/// from the path
/// </summary>
/// <param name="path">Path to chunk and obtain information</param>
/// <param name="tableName">Name of the table as represented in the
/// path</param>
/// <param name="rowNumber">Row number obtained from the path</param>
/// <returns>what the path represents</returns>
private PathType GetNamesFromPath(string path, out string tableName,
out int rowNumber)
{
    PathType retVal = PathType.Invalid;
    rowNumber = -1;
    tableName = null;

    // Check if the path specified is a drive
    if (PathIsDrive(path))
    {
        return PathType.Database;
    }

    // chunk the path into parts
    string[] pathChunks = ChunkPath(path);

    switch (pathChunks.Length)

```

```

    {
        case 1:
        {
            string name = pathChunks[0];

            if (TableNameIsValid(name))
            {
                tableName = name;
                retVal = PathType.Table;
            }
        }
        break;

        case 2:
        {
            string name = pathChunks[0];

            if (TableNameIsValid(name))
            {
                tableName = name;
            }

            int number = SafeConvertRowNumber(pathChunks[1]);

            if (number >= 0)
            {
                rowNumber = number;
                retVal = PathType.Row;
            }
            else
            {
                WriteError(new ErrorRecord(
                    new ArgumentException("Row number is not
valid"),
                    "RowNumberNotValid",
                    ErrorCategory.InvalidArgument,
                    path));
            }
        }
        break;

        default:
        {
            WriteError(new ErrorRecord(
                new ArgumentException("The path supplied has too
many segments"),
                "PathNotValid",
                ErrorCategory.InvalidArgument,
                path));
        }
        break;
    } // switch(pathChunks...

    return retVal;
} // GetNamesFromPath

```

```

/// <summary>
/// Throws an argument exception stating that the specified path does
/// not represent either a table or a row
/// </summary>
/// <param name="path">path which is invalid</param>
private void ThrowTerminatingInvalidPathException(string path)
{
    StringBuilder message = new StringBuilder("Path must represent
either a table or a row :");
    message.Append(path);

    throw new ArgumentException(message.ToString());
}

/// <summary>
/// Retrieve the list of tables from the database.
/// </summary>
/// <returns>
/// Collection of DatabaseTableInfo objects, each object representing
/// information about one database table
/// </returns>
private Collection<DatabaseTableInfo> GetTables()
{
    Collection<DatabaseTableInfo> results =
        new Collection<DatabaseTableInfo>();

    // using ODBC connection to the database and get the schema of
    tables
    AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

    if (di == null)
    {
        return null;
    }

    OdbcConnection connection = di.Connection;
    DataTable dt = connection.GetSchema("Tables");
    int count;

    // iterate through all rows in the schema and create
    DatabaseTableInfo
    // objects which represents a table
    foreach (DataRow dr in dt.Rows)
    {
        String tableName = dr["TABLE_NAME"] as String;
        DataColumnCollection columns = null;

        // find the number of rows in the table
        try
        {
            String cmd = "Select count(*) from \\" + tableName + "\\";
            OdbcCommand command = new OdbcCommand(cmd, connection);

            count = (Int32)command.ExecuteScalar();
        }
    }
}

```

```

        }

        catch
        {
            count = 0;
        }

        // create DatabaseTableInfo object representing the table
        DatabaseTableInfo table =
            new DatabaseTableInfo(dr, tableName, count, columns);

        results.Add(table);
    } // foreach (DataRow...

    return results;
} // GetTables

/// <summary>
/// Return row information from a specified table.
/// </summary>
/// <param name="tableName">The name of the database table from
/// which to retrieve rows.</param>
/// <returns>Collection of row information objects.</returns>
private Collection<DatabaseRowInfo> GetRows(string tableName)
{
    Collection<DatabaseRowInfo> results =
        new Collection<DatabaseRowInfo>();

    // Obtain rows in the table and add it to the collection
    try
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);

        if (da == null)
        {
            return null;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        int i = 0;
        foreach (DataRow row in table.Rows)
        {
            results.Add(new DatabaseRowInfo(row,
i.ToString(CultureInfo.CurrentCulture)));
            i++;
        } // foreach (DataRow...
    }
    catch (Exception e)
    {
        WriteError(new ErrorRecord(e, "CannotAccessSpecifiedRows",
ErrorCategory.InvalidOperation, tableName));
    }

    return results;
}

```

```

} // GetRows

/// <summary>
/// Retrieve information about a single table.
/// </summary>
/// <param name="tableName">The table for which to retrieve
/// data.</param>
/// <returns>Table information.</returns>
private DatabaseTableInfo GetTable(string tableName)
{
    foreach (DatabaseTableInfo table in GetTables())
    {
        if (String.Equals(tableName, table.Name,
StringComparison.OrdinalIgnoreCase))
        {
            return table;
        }
    }

    return null;
} // GetTable

/// <summary>
/// Removes the specified table from the database
/// </summary>
/// <param name="tableName">Name of the table to remove</param>
private void RemoveTable(string tableName)
{
    // validate if tablename is valid and if table is present
    if (String.IsNullOrEmpty(tableName) ||
!TableNameIsValid(tableName) || !TableIsPresent(tableName))
    {
        return;
    }

    // Execute command using ODBC connection to remove a table
    try
    {
        // delete the table using an sql statement
        string sql = "drop table " + tableName;

        AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

        if (di == null)
        {
            return;
        }
        OdbcConnection connection = di.Connection;

        OdbcCommand cmd = new OdbcCommand(sql, connection);
        cmd.ExecuteScalar();
    }
    catch (Exception ex)

```

```

        {
            WriteError(new ErrorRecord(ex, "CannotRemoveSpecifiedTable",
                ErrorCategory.InvalidOperation, null)
            );
        }
    }

} // RemoveTable

/// <summary>
/// Obtain a data adapter for the specified Table
/// </summary>
/// <param name="tableName">Name of the table to obtain the
/// adapter for</param>
/// <returns>Adapter object for the specified table</returns>
/// <remarks>An adapter serves as a bridge between a DataSet (in
memory
/// representation of table) and the data source</remarks>
private OdbcDataAdapter GetAdapterForTable(string tableName)
{
    OdbcDataAdapter da = null;
    AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;

    if (di == null || !TableNameIsValid(tableName) ||
!TableIsPresent(tableName))
    {
        return null;
    }

    OdbcConnection connection = di.Connection;

    try
    {
        // Create a odbc data adpater. This can be sued to update the
        // data source with the records that will be created here
        // using data sets
        string sql = "Select * from " + tableName;
        da = new OdbcDataAdapter(new OdbcCommand(sql, connection));

        // Create a odbc command builder object. This will create sql
        // commands automatically for a single table, thus
        // eliminating the need to create new sql statements for
        // every operation to be done.
        OdbcCommandBuilder cmd = new OdbcCommandBuilder(da);

        // Set the delete cmd for the table here
        sql = "Delete from " + tableName + " where ID = ?";
        da.DeleteCommand = new OdbcCommand(sql, connection);

        // Specify a DeleteCommand parameter based on the "ID"
        // column
        da.DeleteCommand.Parameters.Add(new OdbcParameter());
        da.DeleteCommand.Parameters[0].SourceColumn = "ID";

        // Create an InsertCommand based on the sql string
        // Insert into "tablename" values (?,?,?)"
        // where
    }
}

```

```

        // ? represents a column in the table. Note that
        // the number of ? will be equal to the number of
        // columns
        DataSet ds = new DataSet();

        da.FillSchema(ds, SchemaType.Source);
        ds.Locale = CultureInfo.InvariantCulture;

        sql = "Insert into " + tableName + " values ( ";
        for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
        {
            sql += "?, ";
        }
        sql = sql.Substring(0, sql.Length - 2);
        sql += ")";
        da.InsertCommand = new OdbcCommand(sql, connection);

        // Create parameters for the InsertCommand based on the
        // captions of each column
        for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
        {
            da.InsertCommand.Parameters.Add(new OdbcParameter());
            da.InsertCommand.Parameters[i].SourceColumn =
                ds.Tables["Table"].Columns[i].Caption;
        }

        // Open the connection if its not already open
        if (connection.State != ConnectionState.Open)
        {
            connection.Open();
        }
    }
    catch (Exception e)
    {
        WriteError(new ErrorRecord(e, "CannotAccessSpecifiedTable",
            ErrorCategory.InvalidOperation, tableName));
    }

    return da;
} // GetAdapterForTable

/// <summary>
/// Gets the DataSet (in memory representation) for the table
/// for the specified adapter
/// </summary>
/// <param name="adapter">Adapter to be used for obtaining
/// the table</param>
/// <param name="tableName">Name of the table for which a
/// DataSet is required</param>
/// <returns>The DataSet with the filled in schema</returns>
private DataSet GetDataSetForTable(OdbcDataAdapter adapter, string
tableName)
{
    Debug.Assert(adapter != null);
}

```

```

// Create a dataset object which will provide an in-memory
// representation of the data being worked upon in the
// data source.
DataSet ds = new DataSet();

// Create a table named "Table" which will contain the same
// schema as in the data source.
//adapter.FillSchema(ds, SchemaType.Source);
adapter.Fill(ds, tableName);
ds.Locale = CultureInfo.InvariantCulture;

return ds;
} //GetDataSetForTable

/// <summary>
/// Get the DataTable object which can be used to operate on
/// for the specified table in the data source
/// </summary>
/// <param name="ds">DataSet object which contains the tables
/// schema</param>
/// <param name="tableName">Name of the table</param>
/// <returns>Corresponding DataTable object representing
/// the table</returns>
///
private DataTable GetDataTable(DataSet ds, string tableName)
{
    Debug.Assert(ds != null);
    Debug.Assert(tableName != null);

    DataTable table = ds.Tables[tableName];
    table.Locale = CultureInfo.InvariantCulture;

    return table;
} // GetDataTable

/// <summary>
/// Retrieves a single row from the named table.
/// </summary>
/// <param name="tableName">The table that contains the
/// numbered row.</param>
/// <param name="row">The index of the row to return.</param>
/// <returns>The specified table row.</returns>
private DatabaseRowInfo GetRow(string tableName, int row)
{
    Collection<DatabaseRowInfo> di = GetRows(tableName);

    // if the row is invalid write an appropriate error else return
    the
    // corresponding row information
    if (row < di.Count && row >= 0)
    {
        return di[row];
    }
    else

```

```

        {
            WriteError(new ErrorRecord(
                new ItemNotFoundException(),
                "RowNotFound",
                ErrorCategory.ObjectNotFound,
                row.ToString(CultureInfo.CurrentCulture)))
        );
    }

    return null;
} // GetRow

/// <summary>
/// Method to safely convert a string representation of a row number
/// into its Int32 equivalent
/// </summary>
/// <param name="rowNumberAsStr">String representation of the row
/// number</param>
/// <remarks>If there is an exception, -1 is returned</remarks>
private int SafeConvertRowNumber(string rowNumberAsStr)
{
    int rowNumber = -1;
    try
    {
        rowNumber = Convert.ToInt32(rowNumberAsStr,
CultureInfo.CurrentCulture);
    }
    catch (FormatException fe)
    {
        WriteError(new ErrorRecord(fe, "RowStringFormatNotValid",
ErrorCategory.InvalidData, rowNumberAsStr));
    }
    catch (OverflowException oe)
    {
        WriteError(new ErrorRecord(oe,
"RowStringConversionToNumberFailed",
ErrorCategory.InvalidData, rowNumberAsStr));
    }
}

return rowNumber;
} // SafeConvertRowNumber

/// <summary>
/// Check if a table name is valid
/// </summary>
/// <param name="tableName">Table name to validate</param>
/// <remarks>Helps to check for SQL injection attacks</remarks>
private bool TableNameIsValid(string tableName)
{
    Regex exp = new Regex(pattern, RegexOptions.Compiled |
RegexOptions.IgnoreCase);

    if (exp.IsMatch(tableName))
    {
        return true;
    }
}

```

```

        }
        WriteError(new ErrorRecord(
            new ArgumentException("Table name not valid"),
            "TableNameNotValid",
                ErrorCategory.InvalidArgument, tableName));
        return false;
    } // TableNameIsValid

    /// <summary>
    /// Checks to see if the specified table is present in the
    /// database
    /// </summary>
    /// <param name="tableName">Name of the table to check</param>
    /// <returns>true, if table is present, false otherwise</returns>
    private bool TableIsPresent(string tableName)
    {
        // using ODBC connection to the database and get the schema of
        tables
        AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;
        if (di == null)
        {
            return false;
        }

        OdbcConnection connection = di.Connection;
        DataTable dt = connection.GetSchema("Tables");

        // check if the specified tableName is available
        // in the list of tables present in the database
        foreach (DataRow dr in dt.Rows)
        {
            string name = dr["TABLE_NAME"] as string;
            if (name.Equals(tableName,
StringComparison.OrdinalIgnoreCase))
            {
                return true;
            }
        }
    }

    WriteError(new ErrorRecord(
        new ArgumentException("Specified Table is not present in
database"), "TableNotAvailable",
            ErrorCategory.InvalidArgument, tableName));

    return false;
} // TableIsPresent

    /// <summary>
    /// Gets the next available ID in the table
    /// </summary>
    /// <param name="table">DataTable object representing the table to
    /// search for ID</param>
    /// <returns>next available id</returns>
    private int GetNextID(DataTable table)
    {

```

```

    int big = 0;

    for (int i = 0; i < table.Rows.Count; i++)
    {
        DataRow row = table.Rows[i];

        int id = (int)row["ID"];

        if (big < id)
        {
            big = id;
        }
    }

    big++;
    return big;
}

#endregion Helper Methods

#region Private Properties

private string pathSeparator = "\\";
private static string pattern = @"^+[a-z]+[0-9]*_*$";

private enum PathType { Database, Table, Row, Invalid };

#endregion Private Properties

} // AccessDBProvider

#endregion AccessDBProvider

#region Helper Classes

#region AccessDBPSDriveInfo

/// <summary>
/// Any state associated with the drive should be held here.
/// In this case, it's the connection to the database.
/// </summary>
internal class AccessDBPSDriveInfo : PSDriveInfo
{
    private OdbcConnection connection;

    /// <summary>
    /// ODBC connection information.
    /// </summary>
    public OdbcConnection Connection
    {
        get { return connection; }
        set { connection = value; }
    }

    /// <summary>

```

```
    ///> Summary: Constructor that takes one argument
    ///> Summary: Drive provided by this provider
    public AccessDBPSDriveInfo(PSDriveInfo driveInfo)
        : base(driveInfo)
    { }

} // class AccessDBPSDriveInfo

#endregion AccessDBPSDriveInfo

#region DatabaseTableInfo

    ///> Summary: Contains information specific to the database table.
    ///> Summary: Similar to the DirectoryInfo class.
    ///> Summary:
    public class DatabaseTableInfo
    {
        ///> Summary: Row from the "tables" schema
        public DataRow Data
        {
            get
            {
                return data;
            }
            set
            {
                data = value;
            }
        }
        private DataRow data;

        ///> Summary: The table name.
        ///> Summary:
        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }
        private String name;

        ///> Summary: The number of rows in the table.
        ///> Summary:
        public int RowCount
```

```

{
    get
    {
        return rowCount;
    }
    set
    {
        rowCount = value;
    }
}
private int rowCount;

/// <summary>
/// The column definitions for the table.
/// </summary>
public DataColumnCollection Columns
{
    get
    {
        return columns;
    }
    set
    {
        columns = value;
    }
}
private DataColumnCollection columns;

/// <summary>
/// Constructor.
/// </summary>
/// <param name="row">The row definition.</param>
/// <param name="name">The table name.</param>
/// <param name="rowCount">The number of rows in the table.</param>
/// <param name="columns">Information on the column tables.</param>
public DatabaseTableInfo(DataRow row, string name, int rowCount,
                         DataColumnCollection columns)
{
    Name = name;
    Data = row;
    RowCount = rowCount;
    Columns = columns;
} // DatabaseTableInfo
} // class DatabaseTableInfo

#endregion DatabaseTableInfo

#region DatabaseRowInfo

/// <summary>
/// Contains information specific to an individual table row.
/// Analogous to the FileInfo class.
/// </summary>
public class DatabaseRowInfo
{

```

```

/// <summary>
/// Row data information.
/// </summary>
public DataRow Data
{
    get
    {
        return data;
    }
    set
    {
        data = value;
    }
}
private DataRow data;

/// <summary>
/// The row index.
/// </summary>
public string RowNumber
{
    get
    {
        return rowNumber;
    }
    set
    {
        rowNumber = value;
    }
}
private string rowNumber;

/// <summary>
/// Constructor.
/// </summary>
/// <param name="row">The row information.</param>
/// <param name="name">The row index.</param>
public DatabaseRowInfo(DataRow row, string name)
{
    RowNumber = name;
    Data = row;
} // DatabaseRowInfo
} // class DatabaseRowInfo

#endregion DatabaseRowInfo

#endregion Helper Classes
}

```

See Also

[Windows PowerShell SDK](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

AccessDbProviderSample06 Code Sample

Article • 09/17/2021

The following code shows the implementation of the Windows PowerShell content provider described in [Creating a Windows PowerShell Content Provider](#). This provider enables the user to manipulate the contents of the items in a data store.

ⓘ Note

You can download the C# source file (AccessDBSampleProvider06.cs) for this provider by using the Microsoft Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory. For more information about other Windows PowerShell provider implementations, see [Designing Your Windows PowerShell Provider](#).

Code Sample

C#

```
using System;
using System.IO;
using System.Data;
using System.Data.Odbc;
using System.Diagnostics;
using System.Collections;
using System.Collections.ObjectModel;
using System.Management.Automation;
using System.Management.Automation.Provider;
using System.Text;
using System.Text.RegularExpressions;
using System.ComponentModel;
using System.Globalization;

namespace Microsoft.Samples.PowerShell.Providers
{
    #region AccessDBProvider

    /// <summary>
    /// This example implements the content methods.
    /// </summary>
    [CmdletProvider("AccessDB", ProviderCapabilities.None)]
```

```

    public class AccessDBProvider : NavigationCmdletProvider,
IContentCmdletProvider
{
    #region Drive Manipulation

        /// <summary>
        /// Create a new drive. Create a connection to the database file
and set
        /// the Connection property in the PSDriveInfo.
        /// </summary>
        /// <param name="drive">
        /// Information describing the drive to add.
        /// </param>
        /// <returns>The added drive.</returns>
    protected override PSDriveInfo NewDrive(PSDriveInfo drive)
    {
        // check if drive object is null
        if (drive == null)
        {
            WriteError(new ErrorRecord(
                new ArgumentNullException("drive"),
                "NullDrive",
                ErrorCategory.InvalidArgument,
                null)
            );
            return null;
        }

        // check if drive root is not null or empty
        // and if its an existing file
        if (String.IsNullOrEmpty(drive.Root) || (File.Exists(drive.Root)
== false))
        {
            WriteError(new ErrorRecord(
                new ArgumentException("drive.Root"),
                "NoRoot",
                ErrorCategory.InvalidArgument,
                drive)
            );
            return null;
        }

        // create a new drive and create an ODBC connection to the new
drive
        AccessDBPSDriveInfo accessDBPSDriveInfo = new
AccessDBPSDriveInfo(drive);

        OdbcConnectionStringBuilder builder = new
OdbcConnectionStringBuilder();

        builder.Driver = "Microsoft Access Driver (*.mdb)";
        builder.Add("DBQ", drive.Root);
    }
}

```

```

        OdbcConnection conn = new
OdbcConnection(builder.ConnectionString);
        conn.Open();
        accessDBPSDriveInfo.Connection = conn;

        return accessDBPSDriveInfo;
    } // NewDrive

    /// <summary>
    /// Removes a drive from the provider.
    /// </summary>
    /// <param name="drive">The drive to remove.</param>
    /// <returns>The drive removed.</returns>
protected override PSDriveInfo RemoveDrive(PSDriveInfo drive)
{
    // check if drive object is null
    if (drive == null)
    {
        WriteError(new ErrorRecord(
            new ArgumentNullException("drive"),
            "NullDrive",
            ErrorCategory.InvalidArgument,
            drive)
        );
    }

    return null;
}

// close ODBC connection to the drive
AccessDBPSDriveInfo accessDBPSDriveInfo = drive as
AccessDBPSDriveInfo;

if (accessDBPSDriveInfo == null)
{
    return null;
}
accessDBPSDriveInfo.Connection.Close();

return accessDBPSDriveInfo;
} // RemoveDrive

#endregion Drive Manipulation

#region Item Methods

    /// <summary>
    /// Retrieves an item using the specified path.
    /// </summary>
    /// <param name="path">The path to the item to return.</param>
protected override void GetItem(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {

```

```

        WriteItemObject(this.PSDriveInfo, path, true);
        return;
    }// if (PathIsDrive...

    // Get table name and row information from the path and do
    // necessary actions
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        DatabaseTableInfo table = GetTable(tableName);
        WriteItemObject(table, path, true);
    }
    else if (type == PathType.Row)
    {
        DatabaseRowInfo row = GetRow(tableName, rowNumber);
        WriteItemObject(row, path, false);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }

} // GetItem

/// <summary>
/// Set the content of a row of data specified by the supplied path
/// parameter.
/// </summary>
/// <param name="path">Specifies the path to the row whose columns
/// will be updated.</param>
/// <param name="values">Comma separated string of values</param>
protected override void SetItem(string path, object values)
{
    // Get type, table name and row number from the path specified
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type != PathType.Row)
    {
        WriteError(new ErrorRecord(new NotSupportedException(
            "SetNotSupported"), "",
            ErrorCategory.InvalidOperation, path));

        return;
    }

    // Get in-memory representation of table
}

```

```

OdbcDataAdapter da = GetAdapterForTable(tableName);

    if (da == null)
    {
        return;
    }
    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    if (rowNumber >= table.Rows.Count)
    {
        // The specified row number has to be available. If not
        // NewItem has to be used to add a new row
        throw new ArgumentException("Row specified is not
available");
    } // if (rowNum...

    string[] colValues = (values as string).Split(',');
}

// set the specified row
DataRow row = table.Rows[rowNumber];

for (int i = 0; i < colValues.Length; i++)
{
    row[i] = colValues[i];
}

// Update the table
if (ShouldProcess(path, "SetItem"))
{
    da.Update(ds, tableName);
}

} // SetItem

/// <summary>
/// Test to see if the specified item exists.
/// </summary>
/// <param name="path">The path to the item to verify.</param>
/// <returns>True if the item is found.</returns>
protected override bool ItemExists(string path)
{
    // check if the path represented is a drive
    if (PathIsDrive(path))
    {
        return true;
    }

    // Obtain type, table name and row number from path
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);
}

```

```

DatabaseTableInfo table = GetTable(tableName);

    if (type == PathType.Table)
    {
        // if specified path represents a table then
DatabaseTableInfo
            // object for the same should exist
            if (table != null)
            {
                return true;
            }
        }
        else if (type == PathType.Row)
        {
            // if specified path represents a row then DatabaseTableInfo
should
            // exist for the table and then specified row number must be
within
            // the maximum row count in the table
            if (table != null && rowNum < table.RowCount)
            {
                return true;
            }
        }
    }

    return false;

} // ItemExists

/// <summary>
/// Test to see if the specified path is syntactically valid.
/// </summary>
/// <param name="path">The path to validate.</param>
/// <returns>True if the specified path is valid.</returns>
protected override bool IsValidPath(string path)
{
    bool result = true;

    // check if the path is null or empty
    if (String.IsNullOrEmpty(path))
    {
        result = false;
    }

    // convert all separators in the path to a uniform one
    path = NormalizePath(path);

    // split the path into individual chunks
    string[] pathChunks = path.Split(pathSeparator.ToCharArray());

    foreach (string pathChunk in pathChunks)
    {
        if (pathChunk.Length == 0)
        {
            result = false;
        }
    }
}

```

```

        }
    }
    return result;
} // IsValidPath

#endregion Item Overloads

#region Container Overloads

/// <summary>
/// Return either the tables in the database or the datarows
/// </summary>
/// <param name="path">The path to the parent</param>
/// <param name="recurse">True to return all child items
recursively.
/// </param>
protected override void GetChildItems(string path, bool recurse)
{
    // If path represented is a drive then the children in the path
are
    // tables. Hence all tables in the drive represented will have
to be
    // returned
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table, path, true);

            // if the specified item exists and recurse has been set
then
            // all child items within it have to be obtained as well
            if (ItemExists(path) && recurse)
            {
                GetChildItems(path + pathSeparator + table.Name,
recurse);
            }
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get the table name, row number and type of path from the
        // path specified
        string tableName;
        int rowCount;

        PathType type = GetNamesFromPath(path, out tableName, out
rowCount);

        if (type == PathType.Table)
        {
            // Obtain all the rows within the table
            foreach (DataRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row, path + pathSeparator +

```

```

row.RowNumber,
                false);
        } // foreach (DatabaseRowInfo...
    }
    else if (type == PathType.Row)
    {
        // In this case the user has directly specified a row,
        hence
        // just give that particular row
        DatabaseRowInfo row = GetRow(tableName, rowNumber);
        WriteItemObject(row, path + pathSeparator +
row.RowNumber,
                false);
    }
    else
    {
        // In this case, the path specified is not valid
        ThrowTerminatingInvalidOperationException(path);
    }
} // else
} // GetChildItems

/// <summary>
/// Return the names of all child items.
/// </summary>
/// <param name="path">The root path.</param>
/// <param name="returnContainers">Not used.</param>
protected override void GetChildNames(string path,
                                      ReturnContainers returnContainers)
{
    // If the path represented is a drive, then the child items are
    // tables. get the names of all the tables in the drive.
    if (PathIsDrive(path))
    {
        foreach (DatabaseTableInfo table in GetTables())
        {
            WriteItemObject(table.Name, path, true);
        } // foreach (DatabaseTableInfo...
    } // if (PathIsDrive...
    else
    {
        // Get type, table name and row number from path specified
        string tableName;
        int rowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

        if (type == PathType.Table)
        {
            // Get all the rows in the table and then write out the
            // row numbers.
            foreach (DatabaseRowInfo row in GetRows(tableName))
            {
                WriteItemObject(row.RowNumber, path, false);
            }
        }
    }
}

```

```

        } // foreach (DatabaseRowInfo...
    }
    else if (type == PathType.Row)
    {
        // In this case the user has directly specified a row,
        hence
        // just give that particular row
        DatabaseRowInfo row = GetRow(tableName, rowNumber);

        WriteItemObject(row.RowNumber, path, false);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }
} // else
} // GetChildNames

/// <summary>
/// Determines if the specified path has child items.
/// </summary>
/// <param name="path">The path to examine.</param>
/// <returns>
/// True if the specified path has child items.
/// </returns>
protected override bool HasChildItems(string path)
{
    if (PathIsDrive(path))
    {
        return true;
    }

    return (ChunkPath(path).Length == 1);
} // HasChildItems

/// <summary>
/// Creates a new item at the specified path.
/// </summary>
///
/// <param name="path">
/// The path to the new item.
/// </param>
///
/// <param name="type">
/// Type for the object to create. "Table" for creating a new table
and
/// "Row" for creating a new row in a table.
/// </param>
///
/// <param name="newValue">
/// Object for creating new instance of a type at the specified
path. For
/// creating a "Table" the object parameter is ignored and for
creating
/// a "Row" the object must be of type string which will contain

```

```

comma
    /// separated values of the rows to insert.
    /// </param>
protected override void NewItem(string path, string type,
                                object newItemValue)
{
    string tableName;
    int rowNumber;

    PathType pt = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (pt == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }

    // Check if type is either "table" or "row", if not throw an
    // exception
    if (!String.Equals(type, "table",
 StringComparison.OrdinalIgnoreCase)
        && !String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
    {
        WriteError(new ErrorRecord
                    (new ArgumentException("Type must be
either a table or row"),
                     "CannotCreateSpecifiedObject",
                     ErrorCategory.InvalidArgument,
                     path
                    )
                );
    }

    throw new ArgumentException("This provider can only create
items of type \"table\" or \"row\"");
}

// Path type is the type of path of the container. So if a drive
// is specified, then a table can be created under it and if a
table
// is specified, then a row can be created under it. For the
sake of
// completeness, if a row is specified, then if the row
specified by
// the path does not exist, a new row is created. However, the
row
// number may not match as the row numbers only get incremented
based
// on the number of rows

if (PathIsDrive(path))
{
    if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
    {

```

```

        // Execute command using ODBC connection to create a
table
        try
        {
            // create the table using an sql statement
            string newTableName = newItemValue.ToString();
            string sql = "create table " + newTableName
                + " (ID INT)";

            // Create the table using the Odbc connection from
the
            // drive.
            AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

            if (di == null)
            {
                return;
            }
            OdbcConnection connection = di.Connection;

            if (ShouldProcess(newTableName, "create"))
            {
                OdbcCommand cmd = new OdbcCommand(sql,
connection);
                cmd.ExecuteScalar();
            }
            catch (Exception ex)
            {
                WriteError(new ErrorRecord(ex,
"CannotCreateSpecifiedTable",
ErrorCategory.InvalidOperation, path)
);
            }
        } // if (String...
        else if (String.Equals(type, "row",
 StringComparison.OrdinalIgnoreCase))
{
    throw new
        ArgumentException("A row cannot be created under a
database, specify a path that represents a Table");
}
}// if (PathIsDrive...
else
{
    if (String.Equals(type, "table",
StringComparison.OrdinalIgnoreCase))
    {
        if (rowNumber < 0)
        {
            throw new
                ArgumentException("A table cannot be created
within another table, specify a path that represents a database");
        }
    }
}

```

```

        else
        {
            throw new
                ArgumentException("A table cannot be created
inside a row, specify a path that represents a database");
        }
    } //if (String.Equals.....
// if path specified is a row, create a new row
else if (String.Equals(type, "row",
StringComparison.OrdinalIgnoreCase))
{
    // The user is required to specify the values to be
inserted
    // into the table in a single string separated by commas
    string value = newItemValue as string;

    if (String.IsNullOrEmpty(value))
    {
        throw new
            ArgumentException("Value argument must have
comma separated values of each column in a row");
    }
    string[] rowValues = value.Split(',');
}

OdbcDataAdapter da = GetAdapterForTable(tableName);

if (da == null)
{
    return;
}

DataSet ds = GetDataSetForTable(da, tableName);
DataTable table = GetDataTable(ds, tableName);

if (rowValues.Length != table.Columns.Count)
{
    string message =
String.Format(CultureInfo.CurrentCulture,
                    "The table has {0} columns and
the value specified must have so many comma separated values",
                    table.Columns.Count);

    throw new ArgumentException(message);
}

if (!Force && (rowNumber >= 0 && rowNumber <
table.Rows.Count))
{
    string message =
String.Format(CultureInfo.CurrentCulture,
                    "The row {0} already exists. To
create a new row specify row number as {1}, or specify path to a table, or
use the -Force parameter",
                    rowNumber,
                    table.Rows.Count);
}

```

```
                throw new ArgumentException(message);
            }

            if (rowNumber > table.Rows.Count)
            {
                string message =
String.Format(CultureInfo.CurrentCulture,
                           "To create a new row specify row
number as {0}, or specify path to a table",
                           table.Rows.Count);

                throw new ArgumentException(message);
            }

            // Create a new row and update the row with the input
            // provided by the user
            DataRow row = table.NewRow();
            for (int i = 0; i < rowValues.Length; i++)
            {
                row[i] = rowValues[i];
            }
            table.Rows.Add(row);

            if (ShouldProcess(tableName, "update rows"))
            {
                // Update the table from memory back to the data
source
                da.Update(ds, tableName);
            }

            } // else if (String...
        } // else ...

    } // NewItem

    /// <summary>
    /// Copies an item at the specified path to the location specified
    /// </summary>
    ///
    /// <param name="path">
    /// Path of the item to copy
    /// </param>
    ///
    /// <param name="copyPath">
    /// Path of the item to copy to
    /// </param>
    ///
    /// <param name="recurse">
    /// Tells the provider to recurse subcontainers when copying
    /// </param>
    ///
protected override void CopyItem(string path, string copyPath, bool
recurse)
{

```

```

        string tableName, copyTableName;
        int rowNumber, copyRowNumber;

        PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);
        PathType copyType = GetNamesFromPath(copyPath, out
copyTableName, out copyRowNumber);

        if (type == PathType.Invalid)
        {
            ThrowTerminatingInvalidPathException(path);
        }

        if (type == PathType.Invalid)
        {
            ThrowTerminatingInvalidPathException(copyPath);
        }

        // Get the table and the table to copy to
        OdbcDataAdapter da = GetAdapterForTable(tableName);
        if (da == null)
        {
            return;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        OdbcDataAdapter cda = GetAdapterForTable(copyTableName);
        if (cda == null)
        {
            return;
        }

        DataSet cds = GetDataSetForTable(cda, copyTableName);
        DataTable copyTable = GetDataTable(cds, copyTableName);

        // if source represents a table
        if (type == PathType.Table)
        {
            // if copyPath does not represent a table
            if (copyType != PathType.Table)
            {
                ArgumentException e = new ArgumentException("Table can
only be copied on to another table location");

                WriteError(new ErrorRecord(e, "PathNotValid",
                    ErrorCategory.InvalidArgument, copyPath));

                throw e;
            }

            // if table already exists then force parameter should be
set
            // to force a copy
        }
    }
}

```

```

if (!Force && GetTable(copyTableName) != null)
{
    throw new ArgumentException("Specified path already
exists");
}

for (int i = 0; i < table.Rows.Count; i++)
{
    DataRow row = table.Rows[i];
    DataRow copyRow = copyTable.NewRow();

    copyRow.ItemArray = row.ItemArray;
    copyTable.Rows.Add(copyRow);
}

} // if (type == ...
// if source represents a row
else
{
    if (copyType == PathType.Row)
    {
        if (!Force && (copyRowNumber < copyTable.Rows.Count))
        {
            throw new ArgumentException("Specified path already
exists.");
        }
    }

    DataRow row = table.Rows[rowNumber];
    DataRow copyRow = null;

    if (copyRowNumber < copyTable.Rows.Count)
    {
        // copy to an existing row
        copyRow = copyTable.Rows[copyRowNumber];
        copyRow.ItemArray = row.ItemArray;
        copyRow[0] = GetNextID(copyTable);
    }
    else if (copyRowNumber == copyTable.Rows.Count)
    {
        // copy to the next row in the table that will
        // be created
        copyRow = copyTable.NewRow();
        copyRow.ItemArray = row.ItemArray;
        copyRow[0] = GetNextID(copyTable);
        copyTable.Rows.Add(copyRow);
    }
    else
    {
        // attempting to copy to a nonexistent row or a row
        // that cannot be created now - throw an exception
        string message =
String.Format(CultureInfo.CurrentCulture,
                "The item cannot be specified to
the copied row. Specify row number as {0}, or specify a path to the table.",

                table.Rows.Count);
    }
}

```

```

                throw new ArgumentException(message);
            }
        }
        else
        {
            // destination path specified represents a table,
            // create a new row and copy the item
            DataRow copyRow = copyTable.NewRow();
            copyRow.ItemArray = table.Rows[rowNumber].ItemArray;
            copyRow[0] = GetNextID(copyTable);
            copyTable.Rows.Add(copyRow);
        }
    }

    if (ShouldProcess(copyTableName, "CopyItems"))
    {
        cda.Update(cds, copyTableName);
    }

} //CopyItem

/// <summary>
/// Removes (deletes) the item at the specified path
/// </summary>
///
/// <param name="path">
/// The path to the item to remove.
/// </param>
///
/// <param name="recurse">
/// True if all children in a subtree should be removed, false if
only
/// the item at the specified path should be removed. Is applicable
/// only for container (table) items. Its ignored otherwise (even if
/// specified).
/// </param>
///
/// <remarks>
/// There are no elements in this store which are hidden from the
user.
/// Hence this method will not check for the presence of the Force
/// parameter
/// </remarks>
///

protected override void RemoveItem(string path, bool recurse)
{
    string tableName;
    int rowNumber = 0;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        // if recurse flag has been specified, delete all the rows
}

```

```
as well

    if (recurse)
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);
        if (da == null)
        {
            return;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        for (int i = 0; i < table.Rows.Count; i++)
        {
            table.Rows[i].Delete();
        }

        if (ShouldProcess(path, "RemoveItem"))
        {
            da.Update(ds, tableName);
            RemoveTable(tableName);
        }
        //if (recurse...
        else
        {
            // Remove the table
            if (ShouldProcess(path, "RemoveItem"))
            {
                RemoveTable(tableName);
            }
        }
    }

    else if (type == PathType.Row)
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);
        if (da == null)
        {
            return;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        table.Rows[rowNumber].Delete();

        if (ShouldProcess(path, "RemoveItem"))
        {
            da.Update(ds, tableName);
        }
    }

    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }
}
```

```

} // RemoveItem

#endregion Container Overloads

#region Navigation

/// <summary>
/// Determine if the path specified is that of a container.
/// </summary>
/// <param name="path">The path to check.</param>
/// <returns>True if the path specifies a container.</returns>
protected override bool IsItemContainer(string path)
{
    if (PathIsDrive(path))
    {
        return true;
    }

    string[] pathChunks = ChunkPath(path);
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        foreach (DatabaseTableInfo ti in GetTables())
        {
            if (string.Equals(ti.Name, tableName,
StringComparison.OrdinalIgnoreCase))
            {
                return true;
            }
        } // foreach (DatabaseTableInfo...
    } // if (pathChunks...

    return false;
} // IsItemContainer

/// <summary>
/// Get the name of the leaf element in the specified path
/// </summary>
///
/// <param name="path">
/// The full or partial provider specific path
/// </param>
///
/// <returns>
/// The leaf element in the path
/// </returns>
protected override string GetChildName(string path)
{
    if (PathIsDrive(path))
    {

```

```

        return path;
    }

    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Table)
    {
        return tableName;
    }
    else if (type == PathType.Row)
    {
        return rowNumber.ToString(CultureInfo.CurrentCulture);
    }
    else
    {
        ThrowTerminatingInvalidOperationException(path);
    }

    return null;
}

/// <summary>
/// Removes the child segment of the path and returns the remaining
/// parent portion
/// </summary>
///
/// <param name="path">
/// A full or partial provider specific path. The path may be to an
/// item that may or may not exist.
/// </param>
///
/// <param name="root">
/// The fully qualified path to the root of a drive. This parameter
/// may be null or empty if a mounted drive is not in use for this
/// operation. If this parameter is not null or empty the result
/// of the method should not be a path to a container that is a
/// parent or in a different tree than the root.
/// </param>
///
/// <returns></returns>

protected override string GetParentPath(string path, string root)
{
    // If root is specified then the path has to contain
    // the root. If not nothing should be returned
    if (!String.IsNullOrEmpty(root))
    {
        if (!path.Contains(root))
        {
            return null;
        }
    }
}

```

```
        }

        return path.Substring(0, path.LastIndexOf(pathSeparator,
 StringComparison.OrdinalIgnoreCase));
    }

    /// <summary>
    /// Joins two strings with a provider specific path separator.
    /// </summary>
    ///
    /// <param name="parent">
    /// The parent segment of a path to be joined with the child.
    /// </param>
    ///
    /// <param name="child">
    /// The child segment of a path to be joined with the parent.
    /// </param>
    ///
    /// <returns>
    /// A string that represents the parent and child segments of the
path
    /// joined by a path separator.
    /// </returns>

protected override string MakePath(string parent, string child)
{
    string result;

    string normalParent = NormalizePath(parent);
    normalParent = RemoveDriveFromPath(normalParent);
    string normalChild = NormalizePath(child);
    normalChild = RemoveDriveFromPath(normalChild);

    if (String.IsNullOrEmpty(normalParent) &&
String.IsNullOrEmpty(normalChild))
    {
        result = String.Empty;
    }
    else if (String.IsNullOrEmpty(normalParent) &&
!String.IsNullOrEmpty(normalChild))
    {
        result = normalChild;
    }
    else if (!String.IsNullOrEmpty(normalParent) &&
String.IsNullOrEmpty(normalChild))
    {
        if (normalParent.EndsWith(pathSeparator,
StringComparison.OrdinalIgnoreCase))
        {
            result = normalParent;
        }
        else
        {
            result = normalParent + pathSeparator;
        }
    }
}
```



```

        // Normalize the paths first
        string normalPath = NormalizePath(path);
        normalPath = RemoveDriveFromPath(normalPath);
        string normalBasePath = NormalizePath(basepath);
        normalBasePath = RemoveDriveFromPath(normalBasePath);

        if (String.IsNullOrEmpty(normalBasePath))
        {
            return normalPath;
        }
        else
        {
            if (!normalPath.Contains(normalBasePath))
            {
                return null;
            }

            return normalPath.Substring(normalBasePath.Length +
pathSeparator.Length);
        }
    }

    /// <summary>
    /// Moves the item specified by the path to the specified
destination
    /// </summary>
    ///
    /// <param name="path">
    /// The path to the item to be moved
    /// </param>
    ///
    /// <param name="destination">
    /// The path of the destination container
    /// </param>

protected override void MoveItem(string path, string destination)
{
    // Get type, table name and rowNumber from the path
    string tableName, destTableName;
    int rowNumber, destRowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    PathType destType = GetNamesFromPath(destination, out
destTableName,
                                         out destRowNumber);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }

    if (destType == PathType.Invalid)
    {

```

```
        ThrowTerminatingInvalidOperationException(destination);
    }

    if (type == PathType.Table)
    {
        ArgumentException e = new ArgumentException("Move not
supported for tables");

        WriteError(new ErrorRecord(e, "MoveNotSupported",
            ErrorCategory.InvalidArgument, path));

        throw e;
    }
    else
    {
        OdbcDataAdapter da = GetAdapterForTable(tableName);
        if (da == null)
        {
            return;
        }

        DataSet ds = GetDataSetForTable(da, tableName);
        DataTable table = GetDataTable(ds, tableName);

        OdbcDataAdapter dda = GetAdapterForTable(destTableName);
        if (dda == null)
        {
            return;
        }

        DataSet dds = GetDataSetForTable(dda, destTableName);
        DataTable destTable = GetDataTable(dds, destTableName);
        DataRow row = table.Rows[rowNumber];

        if (destType == PathType.Table)
        {
            DataRow destRow = destTable.NewRow();

            destRow.ItemArray = row.ItemArray;
        }
        else
        {
            DataRow destRow = destTable.Rows[destRowNumber];

            destRow.ItemArray = row.ItemArray;
        }

        // Update the changes
        if (ShouldProcess(path, "MoveItem"))
        {
            WriteItemObject(row, path, false);
            dda.Update(dds, destTableName);
        }
    }
}
```

```

#endregion Navigation

#region Helper Methods

/// <summary>
/// Checks if a given path is actually a drive name.
/// </summary>
/// <param name="path">The path to check.</param>
/// <returns>
/// True if the path given represents a drive, false otherwise.
/// </returns>
private bool PathIsDrive(string path)
{
    // Remove the drive name and first path separator. If the
    // path is reduced to nothing, it is a drive. Also if its
    // just a drive then there wont be any path separators
    if (String.IsNullOrEmpty(
        path.Replace(this.PSDriveInfo.Root, "")) ||
        String.IsNullOrEmpty(
            path.Replace(this.PSDriveInfo.Root + pathSeparator,
            "")))
    {
        return true;
    }
    else
    {
        return false;
    }
} // PathIsDrive

/// <summary>
/// Breaks up the path into individual elements.
/// </summary>
/// <param name="path">The path to split.</param>
/// <returns>An array of path segments.</returns>
private string[] ChunkPath(string path)
{
    // Normalize the path before splitting
    string normalPath = NormalizePath(path);

    // Return the path with the drive name and first path
    // separator character removed, split by the path separator.
    string pathNoDrive = normalPath.Replace(this.PSDriveInfo.Root
        + pathSeparator, "");

    return pathNoDrive.Split(pathSeparator.ToCharArray());
} // ChunkPath

/// <summary>
/// Adapts the path, making sure the correct path separator
/// character is used.
/// </summary>

```

```

/// <param name="path"></param>
/// <returns></returns>
private string NormalizePath(string path)
{
    string result = path;

    if (!String.IsNullOrEmpty(path))
    {
        result = path.Replace("/", pathSeparator);
    }

    return result;
} // NormalizePath

/// <summary>
/// Ensures that the drive is removed from the specified path
/// </summary>
///
/// <param name="path">Path from which drive needs to be
removed</param>
/// <returns>Path with drive information removed</returns>
private string RemoveDriveFromPath(string path)
{
    string result = path;
    string root;

    if (this.PSDriveInfo == null)
    {
        root = String.Empty;
    }
    else
    {
        root = this.PSDriveInfo.Root;
    }

    if (result == null)
    {
        result = String.Empty;
    }

    if (result.Contains(root))
    {
        result = result.Substring(result.IndexOf(root,
 StringComparison.OrdinalIgnoreCase) + root.Length);
    }

    return result;
}

/// <summary>
/// Chunks the path and returns the table name and the row number
/// from the path
/// </summary>
/// <param name="path">Path to chunk and obtain information</param>
/// <param name="tableName">Name of the table as represented in the

```

```

    /// <param name="path">
    /// <param name="rowNumber">Row number obtained from the
path</param>
    /// <returns>what the path represents</returns>
    public PathType GetNamesFromPath(string path, out string tableName,
out int rowNumber)
{
    PathType retVal = PathType.Invalid;
    rowNumber = -1;
    tableName = null;

    // Check if the path specified is a drive
    if (PathIsDrive(path))
    {
        return PathType.Database;
    }

    // chunk the path into parts
    string[] pathChunks = ChunkPath(path);

    switch (pathChunks.Length)
    {
        case 1:
        {
            string name = pathChunks[0];

            if (TableNameIsValid(name))
            {
                tableName = name;
                retVal = PathType.Table;
            }
            break;

        case 2:
        {
            string name = pathChunks[0];

            if (TableNameIsValid(name))
            {
                tableName = name;
            }

            int number = SafeConvertRowNumber(pathChunks[1]);

            if (number >= 0)
            {
                rowNumber = number;
                retVal = PathType.Row;
            }
            else
            {
                WriteError(new ErrorRecord(
                    new ArgumentException("Row number is not
valid")),

```

```

                    "RowNumberNotValid",
                    ErrorCategory.InvalidArgument,
                    path));
                }
            }
            break;

        default:
        {
            WriteError(new ErrorRecord(
                new ArgumentException("The path supplied has too
many segments"),
                "PathNotValid",
                ErrorCategory.InvalidArgument,
                path)));
        }
        break;
    } // switch(pathChunks...

    return retVal;
} // GetNamesFromPath

/// <summary>
/// Throws an argument exception stating that the specified path
does
/// not represent either a table or a row
/// </summary>
/// <param name="path">path which is invalid</param>
private void ThrowTerminatingInvalidPathException(string path)
{
    StringBuilder message = new StringBuilder("Path must represent
either a table or a row :");
    message.Append(path);

    throw new ArgumentException(message.ToString());
}

/// <summary>
/// Retrieve the list of tables from the database.
/// </summary>
/// <returns>
/// Collection of DatabaseTableInfo objects, each object
representing
/// information about one database table
/// </returns>
internal Collection<DatabaseTableInfo> GetTables()
{
    Collection<DatabaseTableInfo> results =
        new Collection<DatabaseTableInfo>();

    // using ODBC connection to the database and get the schema of
tables
    AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;
}

```

```

        if (di == null)
    {
        return null;
    }

    OdbcConnection connection = di.Connection;
    DataTable dt = connection.GetSchema("Tables");
    int count;

    // iterate through all rows in the schema and create
DatabaseTableInfo
    // objects which represents a table
    foreach (DataRow dr in dt.Rows)
    {
        String tableName = dr["TABLE_NAME"] as String;
        DataColumnCollection columns = null;

        // find the number of rows in the table
        try
        {
            String cmd = "Select count(*) from \\" + tableName +
"\\";";
            OdbcCommand command = new OdbcCommand(cmd, connection);

            count = (Int32)command.ExecuteScalar();
        }
        catch
        {
            count = 0;
        }

        // create DatabaseTableInfo object representing the table
        DatabaseTableInfo table =
            new DatabaseTableInfo(dr, tableName, count,
columns);

        results.Add(table);
    } // foreach (DataRow...

    return results;
} // GetTables

/// <summary>
/// Return row information from a specified table.
/// </summary>
/// <param name="tableName">The name of the database table from
/// which to retrieve rows.</param>
/// <returns>Collection of row information objects.</returns>
public Collection<DatabaseRowInfo> GetRows(string tableName)
{
    Collection<DatabaseRowInfo> results =
        new Collection<DatabaseRowInfo>();

    // Obtain rows in the table and add it to the collection
    try

```

```

{
    OdbcDataAdapter da = GetAdapterForTable(tableName);

    if (da == null)
    {
        return null;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    int i = 0;
    foreach (DataRow row in table.Rows)
    {
        results.Add(new DatabaseRowInfo(row,
i.ToString(CultureInfo.CurrentCulture)));
        i++;
    } // foreach (DataRow...
}
catch (Exception e)
{
    WriteError(new ErrorRecord(e, "CannotAccessSpecifiedRows",
ErrorCategory.InvalidOperation, tableName));
}

return results;
} // GetRows

/// <summary>
/// Retrieve information about a single table.
/// </summary>
/// <param name="tableName">The table for which to retrieve
/// data.</param>
/// <returns>Table information.</returns>
private DatabaseTableInfo GetTable(string tableName)
{
    foreach (DatabaseTableInfo table in GetTables())
    {
        if (String.Equals(tableName, table.Name,
 StringComparison.OrdinalIgnoreCase))
        {
            return table;
        }
    }

    return null;
} // GetTable

/// <summary>
/// Removes the specified table from the database
/// </summary>
/// <param name="tableName">Name of the table to remove</param>
private void RemoveTable(string tableName)
{
}

```

```

        // validate if tablename is valid and if table is present
        if (String.IsNullOrEmpty(tableName) ||
    !TableNameIsValid(tableName) || !TableIsPresent(tableName))
        {
            return;
        }

        // Execute command using ODBC connection to remove a table
        try
        {
            // delete the table using an sql statement
            string sql = "drop table " + tableName;

            AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

            if (di == null)
            {
                return;
            }
            OdbcConnection connection = di.Connection;

            OdbcCommand cmd = new OdbcCommand(sql, connection);
            cmd.ExecuteScalar();
        }
        catch (Exception ex)
        {
            WriteError(new ErrorRecord(ex, "CannotRemoveSpecifiedTable",
                ErrorCategory.InvalidOperation, null)
            );
        }
    }

} // RemoveTable

/// <summary>
/// Obtain a data adapter for the specified Table
/// </summary>
/// <param name="tableName">Name of the table to obtain the
/// adapter for</param>
/// <returns>Adapter object for the specified table</returns>
/// <remarks>An adapter serves as a bridge between a DataSet (in
memory
/// representation of table) and the data source</remarks>
internal OdbcDataAdapter GetAdapterForTable(string tableName)
{
    OdbcDataAdapter da = null;
    AccessDBPSDriveInfo di = this.PSDriveInfo as
AccessDBPSDriveInfo;

    if (di == null || !TableNameIsValid(tableName) ||
!TableIsPresent(tableName))
    {
        return null;
    }
}

```

```
OdbcConnection connection = di.Connection;

try
{
    // Create a odbc data adpater. This can be sued to update
    // the
    // data source with the records that will be created here
    // using data sets
    string sql = "Select * from " + tableName;
    da = new OdbcDataAdapter(new OdbcCommand(sql, connection));

    // Create a odbc command builder object. This will create
    // sql
    // commands automatically for a single table, thus
    // eliminating the need to create new sql statements for
    // every operation to be done.
    OdbcCommandBuilder cmd = new OdbcCommandBuilder(da);

    // Set the delete cmd for the table here
    sql = "Delete from " + tableName + " where ID = ?";
    da.DeleteCommand = new OdbcCommand(sql, connection);

    // Specify a DeleteCommand parameter based on the "ID"
    // column
    da.DeleteCommand.Parameters.Add(new OdbcParameter());
    da.DeleteCommand.Parameters[0].SourceColumn = "ID";

    // Create an InsertCommand based on the sql string
    // Insert into "tablename" values (?,?,?,?) where
    // ? represents a column in the table. Note that
    // the number of ? will be equal to the number of
    // columns
    DataSet ds = new DataSet();
    ds.Locale = CultureInfo.InvariantCulture;

    da.FillSchema(ds, SchemaType.Source);

    sql = "Insert into " + tableName + " values ( ";
    for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
    {
        sql += "?, ";
    }
    sql = sql.Substring(0, sql.Length - 2);
    sql += ")";
    da.InsertCommand = new OdbcCommand(sql, connection);

    // Create parameters for the InsertCommand based on the
    // captions of each column
    for (int i = 0; i < ds.Tables["Table"].Columns.Count; i++)
    {
        da.InsertCommand.Parameters.Add(new OdbcParameter());
        da.InsertCommand.Parameters[i].SourceColumn =
            ds.Tables["Table"].Columns[i].Caption;
    }
}
```

```

        // Open the connection if its not already open
        if (connection.State != ConnectionState.Open)
        {
            connection.Open();
        }
    }
    catch (Exception e)
    {
        WriteError(new ErrorRecord(e, "CannotAccessSpecifiedTable",
            ErrorCategory.InvalidOperation, tableName));
    }

    return da;
} // GetAdapterForTable

/// <summary>
/// Gets the DataSet (in memory representation) for the table
/// for the specified adapter
/// </summary>
/// <param name="adapter">Adapter to be used for obtaining
/// the table</param>
/// <param name="tableName">Name of the table for which a
/// DataSet is required</param>
/// <returns>The DataSet with the filled in schema</returns>
internal DataSet GetDataSetForTable(OdbcDataAdapter adapter, string
tableName)
{
    Debug.Assert(adapter != null);

    // Create a dataset object which will provide an in-memory
    // representation of the data being worked upon in the
    // data source.
    DataSet ds = new DataSet();

    // Create a table named "Table" which will contain the same
    // schema as in the data source.
    //adapter.FillSchema(ds, SchemaType.Source);
    adapter.Fill(ds, tableName);
    ds.Locale = CultureInfo.InvariantCulture;

    return ds;
} //GetDataSetForTable

/// <summary>
/// Get the DataTable object which can be used to operate on
/// for the specified table in the data source
/// </summary>
/// <param name="ds">DataSet object which contains the tables
/// schema</param>
/// <param name="tableName">Name of the table</param>
/// <returns>Corresponding DataTable object representing
/// the table</returns>
///
internal DataTable GetDataTable(DataSet ds, string tableName)

```

```

{
    Debug.Assert(ds != null);
    Debug.Assert(tableName != null);

    DataTable table = ds.Tables[tableName];
    table.Locale = CultureInfo.InvariantCulture;

    return table;
} // GetDataTable

/// <summary>
/// Retrieves a single row from the named table.
/// </summary>
/// <param name="tableName">The table that contains the
/// numbered row.</param>
/// <param name="row">The index of the row to return.</param>
/// <returns>The specified table row.</returns>
private DatabaseRowInfo GetRow(string tableName, int row)
{
    Collection<DatabaseRowInfo> di = GetRows(tableName);

    // if the row is invalid write an appropriate error else return
    the
    // corresponding row information
    if (row < di.Count && row >= 0)
    {
        return di[row];
    }
    else
    {
        WriteError(new ErrorRecord(
            new ItemNotFoundException(),
            "RowNotFound",
            ErrorCategory.ObjectNotFound,
            row.ToString(CultureInfo.CurrentCulture))
        );
    }
}

return null;
} // GetRow

/// <summary>
/// Method to safely convert a string representation of a row number
/// into its Int32 equivalent
/// </summary>
/// <param name="rowNumberAsStr">String representation of the row
/// number</param>
/// <remarks>If there is an exception, -1 is returned</remarks>
private int SafeConvertRowNumber(string rowNumberAsStr)
{
    int rowNumber = -1;
    try
    {
        rowNumber = Convert.ToInt32(rowNumberAsStr,
        CultureInfo.CurrentCulture);
    }
}

```

```

        }
        catch (FormatException fe)
        {
            WriteError(new ErrorRecord(fe, "RowStringFormatNotValid",
                ErrorCategory.InvalidData, rowNumberAsStr));
        }
        catch (OverflowException oe)
        {
            WriteError(new ErrorRecord(oe,
"RowStringConversionToNumberFailed",
                ErrorCategory.InvalidData, rowNumberAsStr));
        }

        return rowNumber;
    } // 1

    /// <summary>
    /// Check if a table name is valid
    /// </summary>
    /// <param name="tableName">Table name to validate</param>
    /// <remarks>Helps to check for SQL injection attacks</remarks>
    private bool TableNameIsValid(string tableName)
    {
        Regex exp = new Regex(pattern, RegexOptions.Compiled |
RegexOptions.IgnoreCase);

        if (exp.IsMatch(tableName))
        {
            return true;
        }
        WriteError(new ErrorRecord(
            new ArgumentException("Table name not valid"),
"TableNameNotValid",
            ErrorCategory.InvalidArgument, tableName));
        return false;
    } // TableNameIsValid

    /// <summary>
    /// Checks to see if the specified table is present in the
    /// database
    /// </summary>
    /// <param name="tableName">Name of the table to check</param>
    /// <returns>true, if table is present, false otherwise</returns>
    private bool TableIsPresent(string tableName)
    {
        // using ODBC connection to the database and get the schema of
tables
        AccessDBPSDriveInfo di = this.PSDriveInfo as AccessDBPSDriveInfo;
        if (di == null)
        {
            return false;
        }

        OdbcConnection connection = di.Connection;

```

```

        DataTable dt = connection.GetSchema("Tables");

        // check if the specified tableName is available
        // in the list of tables present in the database
        foreach (DataRow dr in dt.Rows)
        {
            string name = dr["TABLE_NAME"] as string;
            if (name.Equals(tableName,
StringComparison.OrdinalIgnoreCase))
            {
                return true;
            }
        }

        WriteError(new ErrorRecord(
            new ArgumentException("Specified Table is not present in
database"), "TableNotAvailable",
            ErrorCategory.InvalidArgument, tableName));

        return false;
    } // TableIsPresent

    /// <summary>
    /// Gets the next available ID in the table
    /// </summary>
    /// <param name="table">DataTable object representing the table to
    /// search for ID</param>
    /// <returns>next available id</returns>
    private int GetNextID(DataTable table)
    {
        int big = 0;

        for (int i = 0; i < table.Rows.Count; i++)
        {
            DataRow row = table.Rows[i];

            int id = (int)row["ID"];

            if (big < id)
            {
                big = id;
            }
        }

        big++;
        return big;
    }
} //endregion Helper Methods

#region Content Methods

    /// <summary>
    /// Clear the contents at the specified location. In this case,
    clearing
    /// the item amounts to clearing a row

```

```

/// </summary>
/// <param name="path">The path to the content to clear.</param>
public void ClearContent(string path)
{
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type != PathType.Table)
    {
        WriteError(new ErrorRecord(
            new InvalidOperationException("Operation not supported.
Content can be cleared only for table"),
            "NotValidRow", ErrorCategory.InvalidArgument,
            path));
        return;
    }

    OdbcDataAdapter da = GetAdapterForTable(tableName);

    if (da == null)
    {
        return;
    }

    DataSet ds = GetDataSetForTable(da, tableName);
    DataTable table = GetDataTable(ds, tableName);

    // Clear contents at the specified location
    for (int i = 0; i < table.Rows.Count; i++)
    {
        table.Rows[i].Delete();
    }

    if (ShouldProcess(path, "ClearContent"))
    {
        da.Update(ds, tableName);
    }
}

} // ClearContent

/// <summary>
/// Not implemented.
/// </summary>
/// <param name="path"></param>
/// <returns></returns>
public object ClearContentDynamicParameters(string path)
{
    return null;
}

/// <summary>
/// Get a reader at the path specified.

```

```
/// </summary>
/// <param name="path">The path from which to read.</param>
/// <returns>A content reader used to read the data.</returns>
public IContentReader GetContentReader(string path)
{
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }
    else if (type == PathType.Row)
    {
        throw new InvalidOperationException("contents can be obtained
only for tables");
    }

    return new AccessDBContentReader(path, this);
} // GetContentReader

/// <summary>
/// Not implemented.
/// </summary>
/// <param name="path"></param>
/// <returns></returns>
public object GetContentReaderDynamicParameters(string path)
{
    return null;
}

/// <summary>
/// Get an object used to write content.
/// </summary>
/// <param name="path">The root path at which to write.</param>
/// <returns>A content writer for writing.</returns>
public IContentWriter GetContentWriter(string path)
{
    string tableName;
    int rowNumber;

    PathType type = GetNamesFromPath(path, out tableName, out
rowNumber);

    if (type == PathType.Invalid)
    {
        ThrowTerminatingInvalidPathException(path);
    }
    else if (type == PathType.Row)
    {
        throw new InvalidOperationException("contents can be added
only to tables");
    }
}
```

```

        }

        return new AccessDBContentWriter(path, this);
    }

    /// <summary>
    /// Not implemented.
    /// </summary>
    /// <param name="path"></param>
    /// <returns></returns>
    public object GetContentWriterDynamicParameters(string path)
    {
        return null;
    }

#endregion Content Methods

#region Private Properties

private string pathSeparator = "\\";
private static string pattern = @"^+[a-z]+[0-9]*_*$";

#endregion Private Properties

} // AccessDBProvider

#endregion AccessDBProvider

#region Helper Classes

#region Public Enumerations

/// <summary>
/// Type of item represented by the path
/// </summary>
public enum PathType
{
    /// <summary>
    /// Represents a database
    /// </summary>
    Database,
    /// <summary>
    /// Represents a table
    /// </summary>
    Table,
    /// <summary>
    /// Represents a row
    /// </summary>
    Row,
    /// <summary>
    /// Represents an invalid path
    /// </summary>
    Invalid
};


```

```

#endifregion Public Enumerations

#region AccessDBPSDriveInfo

/// <summary>
/// Any state associated with the drive should be held here.
/// In this case, it's the connection to the database.
/// </summary>
internal class AccessDBPSDriveInfo : PSDriveInfo
{
    private OdbcConnection connection;

    /// <summary>
    /// ODBC connection information.
    /// </summary>
    public OdbcConnection Connection
    {
        get { return connection; }
        set { connection = value; }
    }

    /// <summary>
    /// Constructor that takes one argument
    /// </summary>
    /// <param name="driveInfo">Drive provided by this provider</param>
    public AccessDBPSDriveInfo(PSDriveInfo driveInfo)
        : base(driveInfo)
    {
    }

} // class AccessDBPSDriveInfo

#endifregion AccessDBPSDriveInfo

#region DatabaseTableInfo

/// <summary>
/// Contains information specific to the database table.
/// Similar to the DirectoryInfo class.
/// </summary>
public class DatabaseTableInfo
{
    /// <summary>
    /// Row from the "tables" schema
    /// </summary>
    public DataRow Data
    {
        get
        {
            return data;
        }
        set
        {
            data = value;
        }
    }
}

```

```
private DataRow data;

/// <summary>
/// The table name.
/// </summary>
public string Name
{
    get
    {
        return name;
    }
    set
    {
        name = value;
    }
}
private String name;

/// <summary>
/// The number of rows in the table.
/// </summary>
public int RowCount
{
    get
    {
        return rowCount;
    }
    set
    {
        rowCount = value;
    }
}
private int rowCount;

/// <summary>
/// The column definitions for the table.
/// </summary>
public DataColumnCollection Columns
{
    get
    {
        return columns;
    }
    set
    {
        columns = value;
    }
}
private DataColumnCollection columns;

/// <summary>
/// Constructor.
/// </summary>
/// <param name="row">The row definition.</param>
/// <param name="name">The table name.</param>
```

```
/// <param name="rowCount">The number of rows in the table.</param>
/// <param name="columns">Information on the column tables.</param>
public DatabaseTableInfo(DataRow row, string name, int rowCount,
                         DataColumnCollection columns)
{
    Name = name;
    Data = row;
    RowCount = rowCount;
    Columns = columns;
} // DatabaseTableInfo
} // class DatabaseTableInfo

#endregion DatabaseTableInfo

#region DatabaseRowInfo

/// <summary>
/// Contains information specific to an individual table row.
/// Analogous to the FileInfo class.
/// </summary>
public class DatabaseRowInfo
{
    /// <summary>
    /// Row data information.
    /// </summary>
    public DataRow Data
    {
        get
        {
            return data;
        }
        set
        {
            data = value;
        }
    }
    private DataRow data;

    /// <summary>
    /// The row index.
    /// </summary>
    public string RowNumber
    {
        get
        {
            return RowNumber;
        }
        set
        {
            RowNumber = value;
        }
    }
    private string RowNumber;

    /// <summary>
```

```

///<summary>
///The row information.
///The row index.
public DatabaseRowInfo(DataRow row, string name)
{
    RowNumber = name;
    Data = row;
} // DatabaseRowInfo
} // class DatabaseRowInfo

#endregion DatabaseRowInfo

#region AccessDBContentReader

///<summary>
///Content reader used to retrieve data from this provider.
///</summary>
public class AccessDBContentReader : IContentReader
{
    // A provider instance is required so as to get "content"
    private AccessDBProvider provider;
    private string path;
    private long currentOffset;

    internal AccessDBContentReader(string path, AccessDBProvider provider)
    {
        this.path = path;
        this.provider = provider;
    }

    ///<summary>
    ///Read the specified number of rows from the source.
    ///</summary>
    ///<param name="readCount">The number of items to
    ///return.</param>
    ///<returns>An array of elements read.</returns>
    public IList Read(long readCount)
    {
        // Read the number of rows specified by readCount and increment
        // offset
        string tableName;
        int rowNumber;
        PathType type = provider.GetNamesFromPath(path, out tableName,
out rowNumber);

        Collection<DatabaseRowInfo> rows =
            provider.GetRows(tableName);
        Collection<DataRow> results = new Collection<DataRow>();

        if (currentOffset < 0 || currentOffset >= rows.Count)
        {
            return null;
        }
    }
}

```

```

        int rowsRead = 0;

        while (rowsRead < readCount && currentOffset < rows.Count)
        {
            results.Add(rows[(int)currentOffset].Data);
            rowsRead++;
            currentOffset++;
        }

        return results;
    } // Read

    /// <summary>
    /// Moves the content reader specified number of rows from the
    /// origin
    /// </summary>
    /// <param name="offset">Number of rows to offset</param>
    /// <param name="origin">Starting row from which to offset</param>
    public void Seek(long offset, System.IO.SeekOrigin origin)
    {
        // get the number of rows in the table which will help in
        // calculating current position
        string tableName;
        int rowNumber;

        PathType type = provider.GetNamesFromPath(path, out tableName,
out rowNumber);

        if (type == PathType.Invalid)
        {
            throw new ArgumentException("Path specified must represent a
table or a row :" + path);
        }

        if (type == PathType.Table)
        {
            Collection<DatabaseRowInfo> rows =
provider.GetRows(tableName);

            int numRows = rows.Count;

            if (offset > rows.Count)
            {
                throw new
                    ArgumentException(
                        "Offset cannot be greater than the number of
rows available"
                    );
            }

            if (origin == System.IO.SeekOrigin.Begin)
            {
                // starting from Beginning with an index 0, the current
                offset

```

```

                // has to be advanced to offset - 1
                currentOffset = offset - 1;
            }
            else if (origin == System.IO.SeekOrigin.End)
            {
                // starting from the end which is numRows - 1, the
                current
                // offset is so much less than numRows - 1
                currentOffset = numRows - 1 - offset;
            }
            else
            {
                // calculate from the previous value of current offset
                // advancing forward always
                currentOffset += offset;
            }
        } // if (type...
        else
        {
            // for row, the offset will always be set to 0
            currentOffset = 0;
        }
    }

} // Seek

/// <summary>
/// Closes the content reader, so all members are reset
/// </summary>
public void Close()
{
    Dispose();
} // Close

/// <summary>
/// Dispose any resources being used
/// </summary>
public void Dispose()
{
    Seek(0, System.IO.SeekOrigin.Begin);

    GC.SuppressFinalize(this);
} // Dispose
} // AccessDBContentReader

#endregion AccessDBContentReader

#region AccessDBContentWriter

/// <summary>
/// Content writer used to write data in this provider.
/// </summary>
public class AccessDBContentWriter : IContentWriter
{
    // A provider instance is required so as to get "content"
    private AccessDBProvider provider;
}

```

```

    private string path;
    private long currentOffset;

    internal AccessDBContentWriter(string path, AccessDBProvider
provider)
    {
        this.path = path;
        this.provider = provider;
    }

    /// <summary>
    /// Write the specified row contents in the source
    /// </summary>
    /// <param name="content"> The contents to be written to the source.
    /// </param>
    /// <returns>An array of elements which were successfully written to
    /// the source</returns>
    ///
    public IList Write(IList content)
    {
        if (content == null)
        {
            return null;
        }

        // Get the total number of rows currently available it will
        // determine how much to overwrite and how much to append at
        // the end
        string tableName;
        int rowNumber;
        PathType type = provider.GetNamesFromPath(path, out tableName,
out rowNumber);

        if (type == PathType.Table)
        {
            OdbcDataAdapter da = provider.GetAdapterForTable(tableName);
            if (da == null)
            {
                return null;
            }

            DataSet ds = provider.GetDataSetForTable(da, tableName);
            DataTable table = provider.GetDataTable(ds, tableName);

            string[] colValues = (content[0] as string).Split(',');
            // set the specified row
            DataRow row = table.NewRow();

            for (int i = 0; i < colValues.Length; i++)
            {
                if (!String.IsNullOrEmpty(colValues[i]))
                {
                    row[i] = colValues[i];
                }
            }
        }
    }
}

```

```

        }

        //table.Rows.InsertAt(row, rowNumber);
        // Update the table
        table.Rows.Add(row);
        da.Update(ds, tableName);

    }
else
{
    throw new InvalidOperationException("Operation not
supported. Content can be added only for tables");
}

return null;
} // Write

/// <summary>
/// Moves the content reader specified number of rows from the
/// origin
/// </summary>
/// <param name="offset">Number of rows to offset</param>
/// <param name="origin">Starting row from which to offset</param>
public void Seek(long offset, System.IO.SeekOrigin origin)
{
    // get the number of rows in the table which will help in
    // calculating current position
    string tableName;
    int rowNumber;

    PathType type = provider.GetNamesFromPath(path, out tableName,
out rowNumber);

    if (type == PathType.Invalid)
    {
        throw new ArgumentException("Path specified should represent
either a table or a row : " + path);
    }

    Collection<DatabaseRowInfo> rows =
        provider.GetRows(tableName);

    int numRows = rows.Count;

    if (offset > rows.Count)
    {
        throw new
            ArgumentException(
                "Offset cannot be greater than the number of rows
available"
            );
    }

    if (origin == System.IO.SeekOrigin.Begin)
    {

```

```

        // starting from Beginning with an index 0, the current
offset
        // has to be advanced to offset - 1
        currentOffset = offset - 1;
    }
    else if (origin == System.IO.SeekOrigin.End)
    {
        // starting from the end which is numRows - 1, the current
        // offset is so much less than numRows - 1
        currentOffset = numRows - 1 - offset;
    }
    else
    {
        // calculate from the previous value of current offset
        // advancing forward always
        currentOffset += offset;
    }

} // Seek

/// <summary>
/// Closes the content reader, so all members are reset
/// </summary>
public void Close()
{
    Dispose();
} // Close

/// <summary>
/// Dispose any resources being used
/// </summary>
public void Dispose()
{
    Seek(0, System.IO.SeekOrigin.Begin);

    GC.SuppressFinalize(this);
} // Dispose
} // AccessDBContentWriter

#endregion AccessDBContentWriter

#region Helper Classes
} // namespace Microsoft.Samples.PowerShell.Providers

```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

GetProc01 Code Samples

Article • 09/17/2021

Here are the code samples for the GetProc01 sample cmdlet. This is the basic `Get-Process` cmdlet sample described in [Creating Your First Cmdlet](#). A `Get-Process` cmdlet is designed to retrieve information about all the processes running on the local computer.

For complete sample code, see the following topics.

[] Expand table

Language	Topic
C#	GetProc01 (C#) Sample Code
VB.NET	GetProc01 (VB.NET) Sample Code

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

GetProc01 (C#) Sample Code

Article • 09/17/2021

The following code shows the implementation of the GetProc01 sample cmdlet. Notice that the cmdlet is simplified by leaving the actual work of process retrieval to the [System.Diagnostics.Process.Getprocesses*](#) method.

ⓘ Note

You can download the C# source file (getproc01.cs) for this Get-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory.

Code Sample

C#

```
using System;
using System.Diagnostics;
using System.Management.Automation; //Windows PowerShell
namespace

using System.ComponentModel;

// This sample shows how to create a simple cmdlet. To test this
// cmdlet, the snapin must be registered. First, run the command:
//     installutil GetProcessSample01.dll
// Then run:
//     Add-PSSnapin GetProcessSnapIn01
// After the snapin has been loaded, you can run:
//     get-proc

namespace Microsoft.Samples.PowerShell.Commands
{
    #region GetProcCommand

    /// <summary>
    /// This class implements the Get-Proc cmdlet.
    /// </summary>
    [Cmdlet(VerbsCommon.Get, "Proc")]
    public class GetProcCommand : Cmdlet
    {
        #region Cmdlet Overrides
```

```

/// <summary>
/// The ProcessRecord method calls the Process.GetProcesses
/// method to retrieve the processes of the local computer.
/// Then, the WriteObject method writes the associated processes
/// to the pipeline.
/// </summary>
protected override void ProcessRecord()
{
    // Retrieve the current processes.
    Process[] processes = Process.GetProcesses();

    // Write the processes to the pipeline to make them available
    // to the next cmdlet. The second argument (true) tells Windows
    // PowerShell to enumerate the array and to send one process
    // object at a time to the pipeline.
    WriteObject(processes, true);
}

#endregion Overrides

} //GetProcCommand

#endregion GetProcCommand

#region PowerShell snap-in

/// <summary>
/// Create this sample as an PowerShell snap-in
/// </summary>
[RunInstaller(true)]
public class GetProcPSSnapIn01 : PSSnapIn
{
    /// <summary>
    /// Create an instance of the GetProcPSSnapIn01
    /// </summary>
    public GetProcPSSnapIn01()
        : base()
    {

    }

    /// <summary>
    /// Get a name for this PowerShell snap-in. This name will be used in
    /// registering
    /// this PowerShell snap-in.
    /// </summary>
    public override string Name
    {
        get
        {
            return "GetProcPSSnapIn01";
        }
    }

    /// <summary>

```

```
    ///> Vendor information for this PowerShell snap-in.  
    ///> </summary>  
    public override string Vendor  
{  
        get  
        {  
            return "Microsoft";  
        }  
    }  
  
    ///> <summary>  
    ///> Gets resource information for vendor. This is a string of format:  
    ///> resourceBaseName,resourceName.  
    ///> </summary>  
    public override string VendorResource  
{  
        get  
        {  
            return "GetProcPSSnapIn01,Microsoft";  
        }  
    }  
  
    ///> <summary>  
    ///> Description of this PowerShell snap-in.  
    ///> </summary>  
    public override string Description  
{  
        get  
        {  
            return "This is a PowerShell snap-in that includes the get-  
proc cmdlet.";  
        }  
    }  
}  
  
#endregion PowerShell snap-in  
}
```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

GetProc01 (VB.NET) Sample Code

Article • 09/17/2021

The following code shows the implementation of the GetProc01 sample cmdlet. Notice that the cmdlet is simplified by leaving the actual work of process retrieval to the [System.Diagnostics.Process.Getprocesses*](#) method.

ⓘ Note

You can download the C# source file (getproc01.cs) for this Get-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#).

The downloaded source files are available in the <PowerShell Samples> directory.

Code Sample

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

ⓘ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[ⓘ Open a documentation issue](#)

[ⓘ Provide product feedback](#)

GetProc02 Code Samples

Article • 09/17/2021

Here are the code samples for the GetProc02 sample cmdlet. This is the `Get-Process` cmdlet sample described in [Adding Parameters that Process Command-Line Input](#). This `Get-Process` cmdlet retrieves processes based on their name, and then displays information about the processes at the command line.

Note

You can download the C# source file (getproc02.cs) for this Get-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#).

The downloaded source files are available in the <PowerShell Samples> directory.

For complete sample code, see the following topics.

 Expand table

Language	Topic
C#	GetProc02 (C#) Sample Code
VB.NET	GetProc02 (VB.NET) Sample Code

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

GetProc02 (C#) Sample Code

Article • 09/17/2021

The following code shows the implementation of a `Get-Process` cmdlet that accepts command-line input. Notice that this implementation defines a `Name` parameter to allow command-line input, and it uses the `WriteObject(System.Object,System.Boolean)` method as the output mechanism for sending output objects to the pipeline.

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Commands
{
    using System;
    using System.Diagnostics;
    using System.Management.Automation;      // Windows PowerShell namespace.
    #region GetProcCommand

    /// <summary>
    /// This class implements the get-proc cmdlet.
    /// </summary>
    [Cmdlet(VerbsCommon.Get, "Proc")]
    public class GetProcCommand : Cmdlet
    {
        #region Parameters

        /// <summary>
        /// The names of the processes to act on.
        /// </summary>
        private string[] processNames;

        /// <summary>
        /// Gets or sets the list of process names on which
        /// the Get-Proc cmdlet will work.
        /// </summary>
        [Parameter(Position = 0)]
        [ValidateNotNullOrEmpty]
        public string[] Name
        {
            get { return this.processNames; }
            set { this.processNames = value; }
        }

        #endregion Parameters

        #region Cmdlet Overrides

        /// <summary>
```

```
/// The ProcessRecord method calls the Process.GetProcesses
/// method to retrieve the processes specified by the Name
/// parameter. Then, the WriteObject method writes the
/// associated processes to the pipeline.
/// </summary>
protected override void ProcessRecord()
{
    // If no process names are passed to the cmdlet, get all
    // processes.
    if (this.processNames == null)
    {
        WriteObject(Process.GetProcesses(), true);
    }
    else
    {
        // If process names are passed to cmdlet, get and write
        // the associated processes.
        foreach (string name in this.processNames)
        {
            WriteObject(Process.GetProcessesByName(name), true);
        }
    } // if (processNames...
} // ProcessRecord

#endregion Cmdlet Overrides
} // End GetProcCommand class.

#endregion GetProcCommand
}
```

See Also

[Windows PowerShell SDK](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

GetProc02 (VB.NET) Sample Code

Article • 09/17/2021

The following code shows the implementation of a `Get-Process` cmdlet that accepts command-line input. Notice that this implementation defines a `Name` parameter to allow command-line input, and it uses the `WriteObject(System.Object,System.Boolean)` method as the output mechanism for sending output objects to the pipeline.

Code Sample

See Also

[Windows PowerShell SDK](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

GetProc03 Code Samples

Article • 09/17/2021

Here are the code samples for the GetProc03 sample cmdlet. This is the `Get-Process` cmdlet sample described in [Adding Parameters that Process Pipeline Input](#). This `Get-Process` cmdlet uses a `Name` parameter that accepts input from a pipeline object, retrieves process information from the local computer based on the supplied names, and then displays information about the processes at the command line.

Note

You can download the C# source file (getprov03.cs) for this Get-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#).

The downloaded source files are available in the <PowerShell Samples> directory.

For complete sample code, see the following topics.

 Expand table

Language	Topic
C#	GetProc03 (C#) Sample Code
VB.NET	GetProc03 (VB.NET) Sample Code

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

GetProc03 (C#) Sample Code

Article • 09/17/2021

The following code shows the implementation of a `Get-Process` cmdlet that can accept pipelined input. This implementation defines a `Name` parameter that accepts pipeline input, retrieves process information from the local computer based on the supplied names, and then uses the `WriteObject(System.Object,System.Boolean)` method as the output mechanism for sending objects to the pipeline.

ⓘ Note

You can download the C# source file (getprov03.cs) for this Get-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory.

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Commands
{
    using System;
    using System.Diagnostics;
    using System.Management.Automation;           // Windows PowerShell
    namespace.
        #region GetProcCommand

        /// <summary>
        /// This class implements the get-proc cmdlet.
        /// </summary>
        [Cmdlet(VerbsCommon.Get, "Proc")]
        public class GetProcCommand : Cmdlet
        {
            #region Parameters

            /// <summary>
            /// The names of the processes to act on.
            /// </summary>
            private string[] processNames;

            /// <summary>
            /// Gets or setsthe list of process names on
            /// which the Get-Proc cmdlet will work.
            /// </summary>
        }
}
```

```

/// </summary>
[Parameter(
    Position = 0,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true)]
[ValidateNotNullOrEmpty]
public string[] Name
{
    get { return this.processNames; }
    set { this.processNames = value; }
}

#endregion Parameters

#region Cmdlet Overrides

/// <summary>
/// The ProcessRecord method calls the Process.GetProcesses
/// method to retrieve the processes specified by the Name
/// parameter. Then, the WriteObject method writes the
/// associated processes to the pipeline.
/// </summary>
protected override void ProcessRecord()
{
    // If no process names are passed to the cmdlet, get all
    // processes.
    if (this.processNames == null)
    {
        WriteObject(Process.GetProcesses(), true);
    }
    else
    {
        // If process names are passed to the cmdlet, get and write
        // the associated processes.
        foreach (string name in this.processNames)
        {
            WriteObject(Process.GetProcessesByName(name), true);
        }
    } // if (processNames ...
} // ProcessRecord
#endregion Overrides
} // End GetProcCommand class.

#endregion GetProcCommand
}

```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

GetProc03 (VB.NET) Sample Code

Article • 09/17/2021

The following code shows the implementation of a `Get-Process` cmdlet that can accept pipelined input. This implementation defines a `Name` parameter that accepts pipeline input, retrieves process information from the local computer based on the supplied names, and then uses the `WriteObject(System.Object,System.Boolean)` method as the output mechanism for sending objects to the pipeline.

Code Sample

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

GetProc04 Code Samples

Article • 09/15/2023

Here are the code samples for the GetProc04 sample cmdlet. This is the `Get-Process` cmdlet sample described in [Adding Non-terminating Error Reporting to Your Cmdlet](#). This `Get-Process` cmdlet calls the `System.Management.Automation.Cmdlet.WriteError` method whenever an invalid operation exception is thrown while retrieving process information.

Note

You can download the C# source file (`getprov04.cs`) for this Get-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#).

The downloaded source files are available in the <PowerShell Samples> directory.

For complete sample code, see the following topics.

 Expand table

Language	Topic
C#	GetProc04 (C#) Sample Code
VB.NET	GetProc04 (VB.NET) Sample Code

See Also

- [Windows PowerShell Programmer's Guide](#)
- [Windows PowerShell SDK](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

GetProc04 (C#) Sample Code

Article • 09/15/2023

The following code shows the implementation of a `Get-Process` cmdlet that reports non-terminating errors. This implementation calls the `System.Management.Automation.Cmdlet.WriteError` method to report non-terminating errors.

ⓘ Note

You can download the C# source file (`getprov04.cs`) for this Get-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the `<PowerShell Samples>` directory.

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Commands
{
    using System;
    using System.Diagnostics;
    using System.Management.Automation;           // Windows PowerShell
namespace.

    #region GetProcCommand

        /// <summary>
        /// This class implements the get-proc cmdlet.
        /// </summary>
        [Cmdlet(VerbsCommon.Get, "Proc")]
        public class GetProcCommand : Cmdlet
    {
        #region Parameters

            /// <summary>
            /// The names of the processes to act on.
            /// </summary>
            private string[] processNames;

            /// <summary>
            /// Gets or sets the list of process names on
            /// which the Get-Proc cmdlet will work.
            /// </summary>
    }
}
```

```

[Parameter(
    Position = 0,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true)]
[ValidateNotNullOrEmpty]
public string[] Name
{
    get { return this.processNames; }
    set { this.processNames = value; }
}

#endregion Parameters

#region Cmdlet Overrides

/// <summary>
/// The ProcessRecord method calls the Process.GetProcesses
/// method to retrieve the processes specified by the Name
/// parameter. Then, the WriteObject method writes the
/// associated processes to the pipeline.
/// </summary>
protected override void ProcessRecord()
{
    // If no process names are passed to cmdlet, get all
    // processes.
    if (this.processNames == null)
    {
        WriteObject(Process.GetProcesses(), true);
    }
    else
    {
        // If process names are passed to the cmdlet, get and write
        // the associated processes.
        // If a nonterminating error occurs while retrieving
processes,
        // call the WriteError method to send an error record to the
        // error stream.
        foreach (string name in this.processNames)
        {
            Process[] processes;

            try
            {
                processes = Process.GetProcessesByName(name);
            }
            catch (InvalidOperationException ex)
            {
                WriteError(new ErrorRecord(
                    ex,
                    "UnableToAccessProcessByName",
                    ErrorCategory.InvalidOperation,
                    name));
                continue;
            }
        }
    }
}

```

```
        WriteObject(processes, true);
    } // foreach (string name...
} // else
} // ProcessRecord

#endregion Overrides
} // End GetProcCommand class.

#endregion GetProcCommand
}
```

See Also

- [Windows PowerShell Programmer's Guide](#)
- [Windows PowerShell SDK](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

GetProc04 (VB.NET) Sample Code

Article • 09/15/2023

The following code shows the implementation of a `Get-Process` cmdlet that reports non-terminating errors. This implementation calls the `System.Management.Automation.Cmdlet.WriteError` method to report non-terminating errors.

ⓘ Note

You can download the C# source file (`getprov04.cs`) for this Get-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#).

The downloaded source files are available in the <PowerShell Samples> directory.

Code Sample

See Also

- [Windows PowerShell Programmer's Guide](#)
- [Windows PowerShell SDK](#)

ⓘ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

ⓘ Open a documentation issue

ⓘ Provide product feedback

GetProc05 Code Samples

Article • 09/15/2023

Here are the code samples for the GetProc05 sample cmdlet. This `Get-Process` cmdlet is similar to the cmdlet described in [Adding Non-terminating Error Reporting to Your Cmdlet](#).

[] Expand table

Language	Topic
C#	GetProc05 (C#) Sample Code
VB.NET	GetProc05 (VB.NET) Sample Code

See Also

- [Windows PowerShell SDK](#)



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

GetProc05 (C#) Sample Code

Article • 09/17/2021

Here is the complete C# code for the GetProc05 sample cmdlet.

C#

```
namespace Microsoft.Samples.PowerShell.Commands
{
    using System;
    using System.Collections.Generic;
    using System.Diagnostics;
    using System.Management.Automation;      // Windows PowerShell namespace.
    using System.Security.Permissions;
    using Win32Exception = System.ComponentModel.Win32Exception;
    #region GetProcCommand

    /// <summary>
    /// This class implements the get-proc cmdlet.
    /// </summary>
    [Cmdlet(VerbsCommon.Get, "Proc",
        DefaultParameterSetName = "ProcessName")]
    public class GetProcCommand : PSCmdlet
    {
        #region Fields
        /// <summary>
        /// The names of the processes to act on.
        /// </summary>
        private string[] processNames;

        /// <summary>
        /// The identifiers of the processes to act on.
        /// </summary>
        private int[] processIds;

        /// <summary>
        /// The process objects to act on.
        /// </summary>
        private Process[] inputObjects;

        #endregion Fields

        #region Parameters

        /// <summary>
        /// Gets or sets the list of process names on
        /// which the Get-Proc cmdlet will work.
        /// </summary>
        [Parameter(
            Position = 0,
            ParameterSetName = "ProcessName",
            ValueFromPipeline = true,
```

```

        ValueFromPipelineByPropertyName = true)]]
[ValidateNotNullOrEmpty]
public string[] Name
{
    get { return this.processNames; }
    set { this.processNames = value; }
}

/// <summary>
/// Gets or sets the list of process identifiers on
/// which the Get-Proc cmdlet will work.
/// </summary>
[Parameter(
    ParameterSetName = "Id",
    Mandatory = true,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true,
    HelpMessage = "The unique id of the process to get.")]
public int[] Id
{
    get { return this.processIds; }
    set { this.processIds = value; }
}

/// <summary>
/// Gets or sets Process objects directly. If the input is a
/// stream of [collection of] Process objects, the cmdlet bypasses the
/// ProcessName and Id parameters and reads the Process objects
/// directly. This allows the cmdlet to deal with processes that have
/// wildcard characters in their name.
/// <value>Process objects</value>
/// </summary>
[Parameter(
    ParameterSetName = "InputObject",
    Mandatory = true,
    ValueFromPipeline = true)]
public Process[] Input
{
    get { return this.inputObjects; }
    set { this.inputObjects = value; }
}

#endregion Parameters

#region Cmdlet Overrides

/// <summary>
/// The ProcessRecord method calls the Process.GetProcesses
/// method to retrieve the processes. Then, the WriteObject
/// method writes the associated processes to the pipeline.
/// </summary>
protected override void ProcessRecord()
{
    List<Process> matchingProcesses;
}

```

```

        WriteDebug("Obtaining the list of matching process objects.");

        switch (ParameterSetName)
        {
            case "Id":
                matchingProcesses = this.GetMatchingProcessesById();
                break;
            case "ProcessName":
                matchingProcesses = this.GetMatchingProcessesByName();
                break;
            case "InputObject":
                matchingProcesses = this.GetProcessesByInput();
                break;
            default:
                ThrowTerminatingError(
                    new ErrorRecord(
                        new ArgumentException("Bad ParameterSetName"),
                        "UnableToAccessProcessList",
                        ErrorCategory.InvalidOperation,
                        null));
                return;
        } // switch (ParameterSetName)

        WriteDebug("Outputting the matching process objects.");

        matchingProcesses.Sort(ProcessComparison);

        foreach (Process process in matchingProcesses)
        {
            WriteObject(process);
        }
    } // ProcessRecord

#endregion Overrides

#region protected Methods and Data

/// <summary>
/// Retrieves the list of all processes matching the ProcessName
/// parameter and generates a nonterminating error for each
/// specified process name which is not found even though the name
/// contains no wildcards.
/// </summary>
/// <returns>The matching processes.</returns>
[EnvironmentPermissionAttribute(
    SecurityAction.LinkDemand,
    Unrestricted = true)]
private List<Process> GetMatchingProcessesByName()
{
    new EnvironmentPermission(
        PermissionState.Unrestricted).Assert();

    List<Process> allProcesses =
        new List<Process>(Process.GetProcesses());
}

```

```

// The keys dictionary is used for rapid lookup of
// processes that are already in the matchingProcesses list.
Dictionary<int, byte> keys = new Dictionary<int, byte>();

List<Process> matchingProcesses = new List<Process>();

if (null == this.processNames)
{
    matchingProcesses.AddRange(allProcesses);
}
else
{
    foreach (string pattern in this.processNames)
    {
        WriteVerbose("Finding matches for process name \""
            + pattern + "\".");
    }

    // WildCard serach on the available processes
    WildcardPattern wildcard =
        new WildcardPattern(
            pattern,
            WildcardOptions.IgnoreCase);

    bool found = false;

    foreach (Process process in allProcesses)
    {
        if (!keys.ContainsKey(process.Id))
        {
            string processName = SafeGetProcessName(process);

            // Remove the process from the allProcesses list
            // so that it is not tested again.
            if (processName.Length == 0)
            {
                allProcesses.Remove(process);
            }

            // Perform a wildcard search on this particular
            // process name and check whether it matches the
            // pattern specified.
            if (!wildcard.IsMatch(processName))
            {
                continue;
            }

            WriteDebug("Found matching process id "
                + process.Id + ".");
        }
    }

    // A match is found.
    found = true;

    // Store the process identifier so that the same
    // process
    // is not added twice.
}

```

```

        keys.Add(process.Id, 0);

        // Add the process to the processes list.
        matchingProcesses.Add(process);
    }
} // foreach (Process...

if (!found &
    !WildcardPattern.ContainsWildcardCharacters(pattern))
{
    WriteError(new ErrorRecord(
        new ArgumentException("Cannot find process name "
            + "\\" + pattern + "\\."),
        "ProcessNameNotFound",
        ErrorCategory.ObjectNotFound,
        pattern));
}
} // foreach (string...
} // if (null...

return matchingProcesses;
} // GetMatchingProcessesByName

/// <summary>
/// Returns the name of a process. If an error occurs, a blank
/// string is returned.
/// </summary>
/// <param name="process">The process whose name is
/// returned.</param>
/// <returns>The name of the process.</returns>
[EnvironmentPermissionAttribute(
    SecurityAction.LinkDemand, Unrestricted = true)]
protected static string SafeGetProcessName(Process process)
{
    new EnvironmentPermission(PermissionState.Unrestricted).Assert();
    string name = String.Empty;

    if (process != null)
    {
        try
        {
            return process.ProcessName;
        }
        catch (Win32Exception)
        {
        }
        catch (InvalidOperationException)
        {
        }
    }
}

return name;
} // SafeGetProcessName

#endregion Cmdlet Overrides

```

```

#region Private Methods

    /// <summary>
    /// Function to sort by process name first, and then by
    /// the process identifier.
    /// </summary>
    /// <param name="x">First process object.</param>
    /// <param name="y">Second process object.</param>
    /// <returns>
    /// Returns less than zero if x is less than y,
    /// greater than 0 if x is greater than y, and 0 if x == y.
    /// </returns>
    private static int ProcessComparison(Process x, Process y)
    {
        int diff = String.Compare(
            SafeGetProcessName(x),
            SafeGetProcessName(y),
            StringComparison.CurrentCultureIgnoreCase);

        if (0 != diff)
        {
            return diff;
        }
        else
        {
            return x.Id.CompareTo(y.Id);
        }
    }

    /// <summary>
    /// Retrieves the list of all processes matching the Id
    /// parameter and generates a nonterminating error for
    /// each specified process identofier which is not found.
    /// </summary>
    /// <returns>
    /// An array of processes that match the given identifier.
    /// </returns>
    [EnvironmentPermissionAttribute(
        SecurityAction.LinkDemand,
        Unrestricted = true)]
    private List<Process> GetMatchingProcessesById()
    {
        new EnvironmentPermission(
            PermissionState.Unrestricted).Assert();

        List<Process> matchingProcesses = new List<Process>();

        if (null != this.processIds)
        {
            // The keys dictionary is used for rapid lookup of the
            // processes already in the matchingProcesses list.
            Dictionary<int, byte> keys = new Dictionary<int, byte>();

            foreach (int processId in this.processIds)

```

```

    {
        WriteVerbose("Finding match for process id "
            + processId + ".");

        if (!keys.ContainsKey(processId))
        {
            Process process;
            try
            {
                process = Process.GetProcessById(processId);
            }
            catch (ArgumentException ex)
            {
                WriteError(new ErrorRecord(
                    ex,
                    "ProcessIdNotFound",
                    ErrorCategory.ObjectNotFound,
                    processId));
                continue;
            }

            WriteDebug("Found matching process.");
            matchingProcesses.Add(process);
            keys.Add(processId, 0);
        }
    }

    return matchingProcesses;
} // GetMatchingProcessesById

/// <summary>
/// Retrieves the list of all processes matching the InputObject
/// parameter.
/// </summary>
/// <returns>The matching processes.</returns>
[EnvironmentPermissionAttribute(
    SecurityAction.LinkDemand,
    Unrestricted = true)]
private List<Process> GetProcessesByInput()
{
    new EnvironmentPermission(
        PermissionState.Unrestricted).Assert();

    List<Process> matchingProcesses = new List<Process>();

    if (null != this.Input)
    {
        // The keys dictionary is used for rapid lookup of the
        // processes already in the matchingProcesses list.
        Dictionary<int, byte> keys = new Dictionary<int, byte>();

        foreach (Process process in this.Input)
        {

```

```
        WriteVerbose("Refreshing process object.");

        if (!keys.ContainsKey(process.Id))
        {
            try
            {
                process.Refresh();
            }
            catch (Win32Exception)
            {
            }
            catch (InvalidOperationException)
            {
            }
        }

        matchingProcesses.Add(process);
    }
}

return matchingProcesses;
} // GetProcessesByInput
#endregion Private Methods
} // End GetProcCommand class.

#endregion GetProcCommand
}
```

See Also

[Windows PowerShell SDK](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

GetProc05 (VB.NET) Sample Code

Article • 12/18/2023

Here is the complete VB.NET code for the GetProc05 sample cmdlet.

```
VB

Imports System
Imports System.Collections.Generic
Imports Win32Exception = System.ComponentModel.Win32Exception
Imports System.Diagnostics
Imports System.Security.Permissions

'Windows PowerShell namespace
Imports System.Management.Automation
Imports System.ComponentModel

Namespace Microsoft.Samples.PowerShell.Commands

    ' This sample is a complete implementation of the Get-Proc Cmdlet.
#Region "GetProcCommand"

    ''' <summary>
    ''' This class implements the Get-Proc cmdlet
    ''' </summary>
<Cmdlet(VerbsCommon.Get, "Proc", _
DefaultParameterSetName:="ProcessName")> _
Public Class GetProcCommand
    Inherits PSCmdlet
#Region "Parameters"

    ''' <summary>
    ''' The list of process names on which this cmdlet will work
    ''' </summary>
<Parameter(Position:=0, ParameterSetName:="ProcessName", _
ValueFromPipeline:=True, _
ValueFromPipelineByPropertyName:=True), _
ValidateNotNullOrEmpty()> _
    Public Property Name() As String()
        Get
            Return processNames
        End Get

        Set(ByVal value As String())
            processNames = value
        End Set
    End Property

    ''' <summary>
    ''' gets/sets an array of process IDs
    ''' </summary>
```

```

<Parameter(ParameterSetName:="Id", _
Mandatory:=True, ValueFromPipeline:=True, _
ValueFromPipelineByPropertyName:=True, _
HelpMessage:="The unique id of the process to get.")> _
Public Property Id() As Integer()
    Get
        Return processIds
    End Get
    Set(ByVal value As Integer())
        processIds = value
    End Set
End Property

''' <summary>
''' If the input is a stream of [collections of] Process
''' objects, we bypass the ProcessName and Id parameters and
''' read the Process objects directly. This allows us to deal
''' with processes which have wildcard characters in their name.
''' <value>Process objects</value>
''' </summary>
<Parameter(ParameterSetName:="InputObject", _
Mandatory:=True, ValueFromPipeline:=True)> _
Public Property Input() As Process()
    Get
        Return inputObjects
    End Get
    Set(ByVal value As Process())
        inputObjects = value
    End Set
End Property

#End Region

#Region "Cmdlet Overrides"

''' <summary>
''' For each of the requested processnames, retrieve and write
''' the associated processes.
''' </summary>
Protected Overrides Sub ProcessRecord()
    Dim matchingProcesses As List(Of Process)

    WriteDebug("Obtaining list of matching process objects.")

    Select Case ParameterSetName
        Case "Id"
            matchingProcesses = GetMatchingProcessesById()
        Case "ProcessName"
            matchingProcesses = GetMatchingProcessesByName()
        Case "InputObject"
            matchingProcesses = GetProcessesByInput()
        Case Else
            ThrowTerminatingError(New ErrorRecord( _
                New ArgumentException("Bad ParameterSetName"), _
                "UnableToAccessProcessList", _

```

```

                ErrorCategory.InvalidOperation, Nothing))
        Return
End Select

WriteDebug("Outputting matching process objects.")

matchingProcesses.Sort(AddressOf ProcessComparison)

Dim process As Process
For Each process In matchingProcesses
    WriteObject(process)
Next process

End Sub 'ProcessRecord

#End Region

#Region "protected Methods and Data"
''' <summary>
''' Retrieves the list of all processes matching the ProcessName
''' parameter.
''' Generates a non-terminating error for each specified
''' process name which is not found even though it contains
''' no wildcards.
''' </summary>
''' <returns></returns>

Private Function GetMatchingProcessesByName() As List(Of Process)

    Dim allProcesses As List(Of Process) = _
        New List(Of Process)(Process.GetProcesses())

    ' The keys dictionary will be used for rapid lookup of
    ' processes already in the matchingProcesses list.
    Dim keys As Dictionary(Of Integer, Byte) = _
        New Dictionary(Of Integer, Byte)()

    Dim matchingProcesses As List(Of Process) = New List(Of Process)
()

If Nothing Is processNames Then
    matchingProcesses.AddRange(allProcesses)
Else
    Dim pattern As String
    For Each pattern In processNames
        WriteVerbose(("Finding matches for process name """ &
            pattern & """."))

        ' WildCard search on the available processes
        Dim wildcard As New WildcardPattern(pattern, _
            WildcardOptions.IgnoreCase)

        Dim found As Boolean = False

        Dim process As Process

```

```

        For Each process In allProcesses
            If Not keys.ContainsKey(process.Id) Then
                Dim processName As String = _
                    SafeGetProcessName(process)

                    ' Remove the process from the allProcesses list
                    ' so that it's not tested again.
                If processName.Length = 0 Then
                    allProcesses.Remove(process)
                End If

                    ' Perform a wildcard search on this particular
                    ' process and check whether this matches the
                    ' pattern specified.
                If Not wildcard.IsMatch(processName) Then
                    GoTo ContinueForEach2
                End If

                    WriteDebug(String.Format( _
                        "Found matching process id """{0}""",

process.Id))

                    ' We have found a match.
                found = True

                    ' Store the process ID so that we don't add the
                    ' same one twice.
                keys.Add(process.Id, 0)

                    ' Add the process to the processes list.
                matchingProcesses.Add(process)
            End If

ContinueForEach2:
    Next process ' foreach (Process...
    If Not found AndAlso Not _
        WildcardPattern.ContainsWildcardCharacters(pattern) _
    Then
        WriteError(New ErrorRecord( _
            New ArgumentException("Cannot find process name
" & _
            " " & pattern & "."), "ProcessNameNotFound", _
            ErrorCategory.ObjectNotFound, pattern))
    End If
    Next pattern
End If
Return matchingProcesses

End Function 'GetMatchingProcessesByName

''' <summary>
''' Returns the name of a process. If an error occurs, a blank
''' string will be returned.
''' </summary>
''' <param name="process">The process whose name will be
''' returned.</param>

```

```

''' <returns>The name of the process.</returns>
Protected Shared Function SafeGetProcessName(ByVal process As
Process) _
    As String

    Dim name As String = ""

    If Not (process Is Nothing) Then
        Try
            Return process.ProcessName
        Catch e1 As Win32Exception
        Catch e2 As InvalidOperationException
        End Try
    End If
    Return name

End Function 'SafeGetProcessName

#End Region

#Region "Private Methods"

''' <summary>
''' Function to sort by ProcessName first, then by Id
''' </summary>
''' <param name="x">first Process object</param>
''' <param name="y">second Process object</param>
''' <returns>
''' returns less than zero if x less than y,
''' greater than 0 if x greater than y, 0 if x == y
''' </returns>
Private Shared Function ProcessComparison(ByVal x As Process, _
    ByVal y As Process) As Integer
    Dim diff As Integer = String.Compare(SafeGetProcessName(x), _
        SafeGetProcessName(y),
        StringComparison.CurrentCultureIgnoreCase)

    If 0 <> diff Then
        Return diff
    End If
    Return x.Id - y.Id
End Function 'ProcessComparison

''' <summary>
''' Retrieves the list of all processes matching the Id
''' parameter.
''' Generates a non-terminating error for each specified
''' process ID which is not found.
''' </summary>
''' <returns>An array of processes that match the given id.
''' </returns>
Protected Function GetMatchingProcessesById() As List(Of Process)

```

```

Dim matchingProcesses As List(Of Process) = New List(Of Process)

If Not (processIds Is Nothing) Then

    ' The keys dictionary will be used for rapid lookup of
    ' processes already in the matchingProcesses list.
    Dim keys As Dictionary(Of Integer, Byte) = _
        New Dictionary(Of Integer, Byte)()

    Dim processId As Integer
    For Each processId In processIds
        WriteVerbose("Finding match for process id " & _
            processId & ".")  
  

        If Not keys.ContainsKey(processId) Then
            Dim process As Process
            Try
                process = _
                    System.Diagnostics.Process.GetProcessById( _
                        processId)
            Catch ex As ArgumentException
                WriteError(New ErrorRecord(ex, _
                    "ProcessIdNotFound", _
                    ErrorCategory.ObjectNotFound, processId))
                GoTo ContinueForEach1
            End Try  
  

            WriteDebug("Found matching process.")

            matchingProcesses.Add(process)
            keys.Add(processId, 0)
        End If
    ContinueForEach1:  

        Next processId
    End If  
  

    Return matchingProcesses
End Function 'GetMatchingProcessesById

''' <summary>
''' Retrieves the list of all processes matching the Input
''' parameter.
''' </summary>
Private Function GetProcessesByInput() As List(Of Process)

    Dim matchingProcesses As List(Of Process) = New List(Of Process)
()

    If Not (Nothing Is Input) Then
        ' The keys dictionary will be used for rapid lookup of
        ' processes already in the matchingProcesses list.
        Dim keys As Dictionary(Of Integer, Byte) = _
            New Dictionary(Of Integer, Byte)()

```

```

        Dim process As Process
        For Each process In Input
            WriteVerbose("Refreshing process object.")

                If Not keys.ContainsKey(process.Id) Then
                    Try
                        process.Refresh()
                    Catch e1 As Win32Exception
                    Catch e2 As InvalidOperationException
                    End Try
                    matchingProcesses.Add(process)
                End If
            Next process
        End If
        Return matchingProcesses
    End Function 'GetProcessesByInput

#End Region

#Region "Private Data"

    Private processNames() As String
    Private processIds() As Integer
    Private inputObjects() As Process

#End Region

End Class 'GetProcCommand

#End Region

#Region "PowerShell snap-in"
    ''' <summary>
    ''' Create this sample as a PowerShell snap-in
    ''' </summary>
    <RunInstaller(True)> _
    Public Class GetProcPSSnapIn05
        Inherits PSSnapIn

        ''' <summary>
        ''' Create an instance of the GetProcPSSnapIn05
        ''' </summary>
        Public Sub New()

            End Sub 'New

            ''' <summary>
            ''' Get a name for this PowerShell snap-in. This name will
            ''' be used in registering
            ''' this PowerShell snap-in.
            ''' </summary>
            Public Overrides ReadOnly Property Name() As String

```

```

    Get
        Return "GetProcPSSnapIn05"
    End Get
End Property

''' <summary>
''' Vendor information for this PowerShell snap-in.
''' </summary>

Public Overrides ReadOnly Property Vendor() As String
    Get
        Return "Microsoft"
    End Get
End Property

''' <summary>
''' Gets resource information for vendor. This is a string of
format:
''' resourceBaseName,resourceName.
''' </summary>

Public Overrides ReadOnly Property VendorResource() As String
    Get
        Return "GetProcPSSnapIn05,Microsoft"
    End Get
End Property

''' <summary>
''' Description of this PowerShell snap-in.
''' </summary>

Public Overrides ReadOnly Property Description() As String
    Get
        Return "This is a PowerShell snap-in that includes " & _
            "the Get-Proc sample."
    End Get
End Property
End Class 'GetProcPSSnapIn05

#End Region

End Namespace

```

See Also

[Windows PowerShell SDK](#)

StopProc01 Code Samples

Article • 09/17/2021

Here is the code sample for the StopProc01 sample cmdlet. This is the `Stop-Process` cmdlet sample described in [Creating a Cmdlet that Modifies the System](#). The `Stop-Process` cmdlet is designed to stop processes that are retrieved using the `Get-Proc` cmdlet (described in [Creating Your First Cmdlet](#)).

Note

You can download the C# (stopproc01.cs) source file for the Stop-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#).

The downloaded source files are available in the <PowerShell Samples> directory.

 Expand table

Language	Topic
C#	StopProc01 (C#) Sample Code

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

StopProc01 (C#) Sample Code

Article • 09/17/2021

Here is the complete C# code for the StopProc01 sample cmdlet.

ⓘ Note

You can download the C# (stopproc01.cs) source file for the Stop-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory.

C#

```
using System;
using System.Diagnostics;
using System.Collections;
using Win32Exception = System.ComponentModel.Win32Exception;
using System.Management.Automation;    // Windows PowerShell namespace
using System.Globalization;

// This sample shows how to implement a PassThru parameter that indicates
// that
// the user wants the cmdlet to return an object, and how to request
// user feedback by calls to the ShouldProcess and ShouldContinue methods.
//
// To test this cmdlet, create a module folder that has the same name as
// this assembly (StopProcesssample01), save the assembly in the module
// folder, and then run the
// following command:
// import-module stopprocesssample01

namespace Microsoft.Samples.PowerShell.Commands
{
    #region StopProcCommand

        /// <summary>
        /// This class implements the stop-proc cmdlet.
        /// </summary>
        [Cmdlet(VerbsLifecycle.Stop, "Proc",
            SupportsShouldProcess = true)]
        public class StopProcCommand : Cmdlet
    {
        #region Parameters

            /// <summary>
            /// This parameter provides the list of process names on
```

```

/// which the Stop-Proc cmdlet will work.
/// </summary>
[Parameter(
    Position = 0,
    Mandatory = true,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true
)]
public string[] Name
{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;

/// <summary>
/// This parameter overrides the ShouldContinue call to force
/// the cmdlet to stop its operation. This parameter should always
/// be used with caution.
/// </summary>
[Parameter]
public SwitchParameter Force
{
    get { return force; }
    set { force = value; }
}
private bool force;

/// <summary>
/// This parameter indicates that the cmdlet should return
/// an object to the pipeline after the processing has been
/// completed.
/// </summary>
[Parameter]
public SwitchParameter PassThru
{
    get { return passThru; }
    set { passThru = value; }
}
private bool passThru;

#endregion Parameters

#region Cmdlet Overrides

/// <summary>
/// The ProcessRecord method does the following for each of the
/// requested process names:
/// 1) Check that the process is not a critical process.
/// 2) Attempt to stop that process.
/// If no process is requested then nothing occurs.
/// </summary>
protected override void ProcessRecord()
{
    foreach (string name in processNames)

```

```

    {
        // For every process name passed to the cmdlet, get the
        associated
        // processes.
        // Write a nonterminating error for failure to retrieve
        // a process.
        Process[] processes;

        try
        {
            processes = Process.GetProcessesByName(name);
        }
        catch (InvalidOperationException ioe)
        {
            WriteError(new
ErrorRecord(ioe, "UnableToAccessProcessByName",
            ErrorCategory.InvalidOperation, name));

            continue;
        }

        // Try to stop the processes that have been retrieved.
        foreach (Process process in processes)
        {
            string processName;

            try
            {
                processName = process.ProcessName;
            }
            catch (Win32Exception e)
            {
                WriteError(new ErrorRecord(e, "ProcessNameNotFound",
                    ErrorCategory.ReadError,
process));
                continue;
            }

            // Confirm the operation with the user first.
            // This is always false if the WhatIf parameter is set.
            if
(!ShouldProcess(string.Format(CultureInfo.CurrentCulture, "{0} ({1})",
processName,
                    process.Id)))
            {
                continue;
            }

            // Make sure that the user really wants to stop a
critical
            // process that could possibly stop the computer.
            bool criticalProcess =
criticalProcessNames.Contains(processName.ToLower(CultureInfo.CurrentCulture
)));

```

```

        if (criticalProcess &&!force)
        {
            string message = String.Format
                (CultureInfo.CurrentCulture,
                 "The process \"\{0}\\" is a critical process
and should not be stopped. Are you sure you wish to stop the process?",

                processName);

            // It is possible that the ProcessRecord method is
            // called
            // multiple times when objects are received as inputs
            // from
            // the pipeline. So to retain YesToAll and NoToAll
            // input that
            // the user may enter across multiple calls to this
            // function,
            // they are stored as private members of the cmdlet.
            if (!ShouldContinue(message, "Warning!",
                                ref yesToAll, ref noToAll))
            {
                continue;
            }
        } // if (criticalProcess...

        // Stop the named process.
        try
        {
            process.Kill();
        }
        catch (Exception e)
        {
            if ((e is Win32Exception) || (e is SystemException)
||

                (e is InvalidOperationException))
            {
                // This process could not be stopped so write
                // a nonterminating error.
                WriteError(new ErrorRecord(e,
"CouldNotStopProcess",
                           ErrorCategory.CloseError,
process));
                continue;
            } // if ((e is...
            else throw;
        } // catch

        // If the PassThru parameter is
        // specified, return the terminated process.
        if (passThru)
        {
            WriteObject(process);
        }
    } // foreach (Process...
} // foreach (string...

```

```
    } // ProcessRecord

    #endregion Cmdlet Overrides

    #region Private Data

    private bool yesToAll, noToAll;

    /// <summary>
    /// Partial list of critical processes that should not be
    /// stopped. Lower case is used for case insensitive matching.
    /// </summary>
    private ArrayList criticalProcessNames = new ArrayList(
        new string[] { "system", "winlogon", "spoolsv" }
    );

    #endregion Private Data

} // StopProcCommand

#endregion StopProcCommand
}
```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

StopProcessSample04 Code Samples

Article • 09/17/2021

Here are the code samples for the StopProc00 sample cmdlet. This is the `Stop-Process` cmdlet sample described in [Adding Parameter Sets to a Cmdlet](#). The `Stop-Process` cmdlet is designed to stop processes that are retrieved using the Get-Proc cmdlet (described in [Creating Your First Cmdlet](#)).

Note

You can download the C# (`stopprocesssample04.cs`) and VB.NET (`stopprocesssample04.vb`) for this Stop-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#).

The downloaded source files are available in the <PowerShell Samples> directory.

For complete sample code, see the following topics.

 Expand table

Language	Topic
C#	StopProc04 (C#) Sample Code
VB.NET	StopProc04 (VB.NET) Sample Code

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

StopProcessSample04 (C#) Sample Code

Article • 09/17/2021

Here is the complete C# sample code for the StopProc04 sample cmdlet. This is the code for the `Stop-Process` cmdlet described in [Adding Parameter Sets to a Cmdlet](#). The `Stop-Process` cmdlet is designed to stop processes that are retrieved using the Get-Proc cmdlet (described in [Creating Your First Cmdlet](#)).

ⓘ Note

You can download the C# (`stopprocesssample04.cs`) source file for this Stop-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the `<PowerShell Samples>` directory.

C#

```
using System;
using System.Diagnostics;
using System.Collections;
using Win32Exception = System.ComponentModel.Win32Exception;
using System.Management.Automation; //Windows PowerShell
namespace
using System.Globalization;

// This sample shows how to declare parameter sets, the input object, and
// how to specify the default parameter set to use.
//
// To test this cmdlet, create a module folder that has the same name as
// this assembly (StopProcesssample04), save the assembly in the module
// folder, and then run the
// following command:
// import-module stopprocesssample04

namespace Microsoft.Samples.PowerShell.Commands
{
    #region StopProcCommand

        /// <summary>
        /// This class implements the stop-proc cmdlet.
        /// </summary>
        [Cmdlet(VerbsLifecycle.Stop, "Proc",
            DefaultParameterSetName = "ProcessId",
            SupportsShouldProcess = true)]
        public class StopProcCommand : PSCmdlet
    {
```

```
#region Parameters

/// <summary>
/// This parameter provides the list of process names on
/// which the Stop-Proc cmdlet will work.
/// </summary>
[Parameter(
    Position = 0,
    ParameterSetName = "ProcessName",
    Mandatory = true,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true,
    HelpMessage = "The name of one or more processes to stop.
Wildcards are permitted."
)]
[Alias("ProcessName")]
public string[] Name
{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;

/// <summary>
/// This parameter overrides the ShouldContinue call to force
/// the cmdlet to stop its operation. This parameter should always
/// be used with caution.
/// </summary>
[Parameter]
public SwitchParameter Force
{
    get { return force; }
    set { force = value; }
}
private bool force;

/// <summary>
/// This parameter indicates that the cmdlet should return
/// an object to the pipeline after the processing has been
/// completed.
/// </summary>
[Parameter(
    HelpMessage = "If set the process(es) will be passed to the
pipeline after stopped."
)]
public SwitchParameter PassThru
{
    get { return passThru; }
    set { passThru = value; }
}
private bool passThru;

/// This parameter provides the list of process identifiers on
/// which the Stop-Proc cmdlet will work.
[Parameter(
```

```

        ParameterSetName = "ProcessId",
        Mandatory = true,
        ValueFromPipelineByPropertyName = true,
        ValueFromPipeline = true
    )]
[Alias("ProcessId")]
public int[] Id
{
    get { return processIds; }
    set { processIds = value; }
}
private int[] processIds;

/// <summary>
/// This parameter accepts an array of Process objects from the
/// the pipeline. This object contains the processes to stop.
/// </summary>
/// <value>Process objects</value>
[Parameter(
    ParameterSetName = "InputObject",
    Mandatory = true,
    ValueFromPipeline = true)]
public Process[] InputObject
{
    get { return inputObject; }
    set { inputObject = value; }
}
private Process[] inputObject;

#endregion Parameters

#region CmdletOverrides

/// <summary>
/// The ProcessRecord method does the following for each of the
/// requested process names:
/// 1) Check that the process is not a critical process.
/// 2) Attempt to stop that process.
/// If no process is requested then nothing occurs.
/// </summary>
protected override void ProcessRecord()
{
    switch (ParameterSetName)
    {
        case "ProcessName":
            ProcessByName();
            break;

        case "ProcessId":
            ProcessById();
            break;

        case "InputObject":
            foreach (Process process in inputObject)
            {

```

```
        SafeStopProcess(process);
    }
    break;

    default:
        throw new ArgumentException("Bad ParameterSet Name");
    } // switch (ParameterSetName...
} // ProcessRecord

#endregion Cmdlet Overrides

#region Helper Methods

/// <summary>
/// Returns all processes with matching names.
/// </summary>
/// <param name="processName">
/// The name of the processes to return.
/// </param>
/// <param name="allProcesses">An array of all
/// computer processes.</param>
/// <returns>An array of matching processes.</returns>
internal ArrayList SafeGetProcessesByName(string processName,
                                         ref ArrayList allProcesses)
{
    // Create and array to store the matching processes.
    ArrayList matchingProcesses = new ArrayList();

    // Create the wildcard for pattern matching.
    WildcardOptions options = WildcardOptions.IgnoreCase |
        WildcardOptions.Compiled;
    WildcardPattern wildcard = new WildcardPattern(processName,
options);

    // Walk all of the machine processes.
    foreach(Process process in allProcesses)
    {
        string processNameToMatch = null;
        try
        {
            processNameToMatch = process.ProcessName;
        }
        catch (Win32Exception e)
        {
            // Remove the process from the list so that it is not
            // checked again.
            allProcesses.Remove(process);

            string message =
                String.Format(CultureInfo.CurrentCulture, "The
process \"{0}\" could not be found",
                processName);
            WriteVerbose(message);
            WriteError(new ErrorRecord(e, "ProcessNotFound",
ErrorCategory.ObjectNotFound,
```

```
processName));  
  
        continue;  
    }  
  
    if (!wildcard.IsMatch(processNameToMatch))  
    {  
        continue;  
    }  
  
    matchingProcesses.Add(process);  
} // foreach(Process...  
  
return matchingProcesses;  
} // SafeGetProcessesByName  
  
/// <summary>  
/// Safely stops a named process. Used as standalone function  
/// to declutter the ProcessRecord method.  
/// </summary>  
/// <param name="process">The process to stop.</param>  
private void SafeStopProcess(Process process)  
{  
    string processName = null;  
  
    try  
    {  
        processName = process.ProcessName;  
    }  
    catch (Win32Exception e)  
    {  
        WriteError(new ErrorRecord(e, "ProcessNotFound",  
            ErrorCategory.OpenError, processName));  
  
        return;  
    }  
  
    // Confirm the operation first.  
    // This is always false if the WhatIf parameter is specified.  
    if (!ShouldProcess(string.Format(CultureInfo.CurrentCulture,  
        "{0} ({1})", processName, process.Id)))  
    {  
        return;  
    }  
  
    // Make sure that the user really wants to stop a critical  
    // process that can possibly stop the computer.  
    bool criticalProcess =  
criticalProcessNames.Contains(processName.ToLower(CultureInfo.CurrentCulture  
));  
  
    string message = null;  
    if (criticalProcess && !force)  
    {  
        message = String.Format(CultureInfo.CurrentCulture,
```



```

processName, process.Id);

WriteVerbose(message);

// If the PassThru parameter is specified, return the terminated
// process to the pipeline.
if (passThru)
{
    // Write a debug message to the host that can be used
    // when troubleshooting a problem. All debug messages
    // will appear with the -Debug option
    message =
        String.Format(CultureInfo.CurrentCulture,
                      "Writing process \'{0}\' to pipeline",
                      processName);
    WriteDebug(message);
    WriteObject(process);
} // if (passThru..
} // SafeStopProcess

/// <summary>
/// Stop processes based on their names (using the
/// ParameterSetName as ProcessName)
/// </summary>
private void ProcessByName()
{
    ArrayList allProcesses = null;

    // Get a list of all processes.
    try
    {
        allProcesses = new ArrayList(Process.GetProcesses());
    }
    catch (InvalidOperationException ioe)
    {
        base.ThrowTerminatingError(new ErrorRecord(
            ioe, "UnableToAccessProcessList",
            ErrorCategory.InvalidOperation, null));
    }

    // If a process name is passed to the cmdlet, get
    // the associated processes.
    // Write a nonterminating error for failure to
    // retrieve a process.
    foreach (string name in processNames)
    {
        // The allProcesses array list is passed as a reference
because
        // any process whose name cannot be obtained will be removed
        // from the list so that its not compared the next time.
        ArrayList processes =
            SafeGetProcessesByName(name, ref allProcesses);

        // If no processes were found write a non-

```

```

        // terminating error.
        if (processes.Count == 0)
        {
            WriteError(new ErrorRecord(
                new Exception("Process not found."),
                "ProcessNotFound",
                ErrorCategory.ObjectNotFound,
                name));
        } // if (processes...
        // Otherwise terminate all processes in the list.
        else
        {
            foreach (Process process in processes)
            {
                SafeStopProcess(process);
            } // foreach (Process...
        } // else
    } // foreach (string...
} // ProcessByName

/// <summary>
/// Stop processes based on their identifiers (using the
/// ParameterSetName as ProcessIds)
/// </summary>
internal void ProcessById()
{
    foreach (int processId in processIds)
    {
        Process process = null;
        try
        {
            process = Process.GetProcessById(processId);

            // Write a debug message to the host that can be used
            // when troubleshooting a problem. All debug messages
            // will appear with the -Debug option
            string message =
                String.Format(CultureInfo.CurrentCulture,
                    "Acquired process for pid : {0}",
                    process.Id);
            WriteDebug(message);
        }
        catch (ArgumentException ae)
        {
            string
                message = String.Format(CultureInfo.CurrentCulture,
                    "The process id {0} could not be
found",
                processId);
            WriteVerbose(message);
            WriteError(new ErrorRecord(ae, "ProcessIdNotFound",
                ErrorCategory.ObjectNotFound,
                processId));
            continue;
        }
    }
}

```

```
        SafeStopProcess(process);
    } // foreach (int...
} // ProcessById

#endregion Helper Methods

#region Private Data

private bool yesToAll, noToAll;

/// <summary>
/// Partial list of critical processes that should not be
/// stopped. Lower case is used for case insensitive matching.
/// </summary>
private ArrayList criticalProcessNames = new ArrayList(
    new string[] { "system", "winlogon", "spoolsv", "calc" })
;

#endregion Private Data

} // StopProcCommand

#endregion StopProcCommand
}
```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:



[Open a documentation issue](#)



[Provide product feedback](#)

StopProcessSample04 (VB.NET) Sample Code

Article • 12/18/2023

Here is the complete VB.NET sample code for the StopProc04 sample cmdlet. This is the code for the `Stop-Process` cmdlet described in [Adding Parameter Sets to a Cmdlet](#). The `Stop-Process` cmdlet is designed to stop processes that are retrieved using the Get-Proc cmdlet (described in [Creating Your First Cmdlet](#)).

ⓘ Note

You can download the VB.NET (stopprocesssample04.vb) source file for this Stop-Proc cmdlet using the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#).

The downloaded source files are available in the <PowerShell Samples> directory.

VB

```
Imports System
Imports System.Diagnostics
Imports System.Collections
Imports Win32Exception = System.ComponentModel.Win32Exception
Imports System.Management.Automation 'Windows PowerShell namespace
Imports System.ComponentModel
Imports System.Globalization

Namespace Microsoft.Samples.PowerShell.Commands

    ' This sample introduces parameter sets, the input object and
    ' DefaultParameterSet.

#Region "StopProcCommand"

    ''' <summary>
    ''' Class that implements the Stop-Proc cmdlet.
    ''' </summary>
    <Cmdlet(VerbsLifecycle.Stop, "Proc",
DefaultParameterSetName:="ProcessId", _
    SupportsShouldProcess:=True)> _
    Public Class StopProcCommand
        Inherits PSCmdlet
#Region "Parameters"
```

```

''' <summary>
''' The list of process names on which this cmdlet will work.
''' </summary>

<Parameter(Position:=0, ParameterSetName:="ProcessName", _
Mandatory:=True, _
ValueFromPipeline:=True, ValueFromPipelineByPropertyName:=True, _
HelpMessage:="The name of one or more processes to stop. " & _
"Wildcards are permitted."), [Alias]("ProcessName")> _
Public Property Name() As String()
    Get
        Return processNames
    End Get
    Set(ByVal value As String())
        processNames = value
    End Set
End Property
Private processNames() As String

''' <summary>
''' Overrides the ShouldContinue check to force stop operation.
''' This option should always be used with caution.
''' </summary>

<Parameter()> _
Public Property Force() As SwitchParameter
    Get
        Return myForce
    End Get
    Set(ByVal value As SwitchParameter)
        myForce = value
    End Set
End Property
Private myForce As Boolean

''' <summary>
''' Common parameter to determine if the process should pass the
''' object down the pipeline after the process has been stopped.
''' </summary>

<Parameter( _
HelpMessage:= _
"If set the process(es) will be passed to the pipeline " & _
"after stopping them.")> _
Public Property PassThru() As SwitchParameter
    Get
        Return myPassThru
    End Get
    Set(ByVal value As SwitchParameter)
        myPassThru = value
    End Set
End Property
Private myPassThru As Boolean

''' <summary>

```

```

''' The list of process IDs on which this cmdlet will work.
''' </summary>

<Parameter(ParameterSetName:="ProcessId", _
Mandatory:=True, _
ValueFromPipelineByPropertyName:=True, _
ValueFromPipeline:=True), [Alias]("ProcessId")> _
Public Property Id() As Integer()
    Get
        Return processIds
    End Get
    Set(ByVal value As Integer())
        processIds = value
    End Set
End Property
Private processIds() As Integer

''' <summary>
''' An array of Process objects from the stream to stop.
''' </summary>
''' <value>Process objects</value>
<Parameter(ParameterSetName:="InputObject", _
Mandatory:=True, ValueFromPipeline:=True)> _
Public Property InputObject() As Process()
    Get
        Return myInputObject
    End Get
    Set(ByVal value As Process())
        myInputObject = value
    End Set
End Property
Private myInputObject() As Process

#End Region

#Region "CmdletOverrides"
''' <summary>
''' For each of the requested processnames:
''' 1) check it's not a special process
''' 2) attempt to stop that process.
''' If no process requested, then nothing occurs.
''' </summary>
Protected Overrides Sub ProcessRecord()
    Select Case ParameterSetName
        Case "ProcessName"
            ProcessByName()

        Case "ProcessId"
            ProcessById()

        Case "InputObject"
            Dim process As Process
            For Each process In myInputObject
                SafeStopProcess(process)
            Next process
    End Select
End Sub

```

```

        Case Else
            Throw New ArgumentException("Bad ParameterSet Name")
        End Select

    End Sub 'ProcessRecord ' ProcessRecord
#End Region

#Region "Helper Methods"
    ''' <summary>
    ''' Returns all processes with matching names.
    ''' </summary>
    ''' <param name="processName">
    ''' The name of the process(es) to return
    ''' </param>
    ''' <param name="allProcesses">An array of all
    ''' machine processes.</param>
    ''' <returns>An array of matching processes.</returns>
    Friend Function SafeGetProcessesByName(ByVal processName As String,
    -
        ByRef allProcesses As ArrayList) As ArrayList

        ' Create and array to store the matching processes.
        Dim matchingProcesses As New ArrayList()

        ' Create the wildcard for pattern matching.
        Dim options As WildcardOptions = WildcardOptions.IgnoreCase Or _
            WildcardOptions.Compiled
        Dim wildcard As New WildcardPattern(processName, options)

        ' Walk all of the machine processes.
        Dim process As Process
        For Each process In allProcesses
            Dim processNameToMatch As String = Nothing
            Try
                processNameToMatch = process.ProcessName
            Catch e As Win32Exception
                ' Remove the process from the list so that it is not
                ' checked again.
                allProcesses.Remove(process)

                Dim message As String = _
                    String.Format(CultureInfo.CurrentCulture, _
                        "The process ""{0}"" could not be found",
processName)
                WriteVerbose(message)
                WriteError(New ErrorRecord(e, _
                    "ProcessNotFound", ErrorCategory.ObjectNotFound, _
                    processName))
            End Try

            If Not wildcard.IsMatch(processNameToMatch) Then
                GoTo ContinueForEach1
            End If
        Next
    End Function

```

```

        End If

        matchingProcesses.Add(process)
ContinueForEach1:
    Next process
    Return matchingProcesses

End Function 'SafeGetProcessesByName

''' <summary>
''' Safely stops a named process. Used as standalone function
''' to declutter ProcessRecord method.
''' </summary>
''' <param name="process">The process to stop.</param>
Private Sub SafeStopProcess(ByVal process As Process)
    Dim processName As String = Nothing

    Try
        processName = process.ProcessName
    Catch e As Win32Exception
        WriteError(New ErrorRecord(e, "ProcessNotFound", _
            ErrorCategory.OpenError, processName))

        Return
    End Try

    ' Confirm the operation first.
    ' This is always false if WhatIf is set.
    If Not ShouldProcess(String.Format(CultureInfo.CurrentCulture, _
        "{0} ({1})", processName, process.Id)) Then
        Return
    End If

    ' Make sure the user really wants to stop a critical
    ' process and possibly stop the machine.
    Dim criticalProcess As Boolean = _
        criticalProcessNames.Contains( _
        processName.ToLower(CultureInfo.CurrentCulture))

    Dim message As String = Nothing
    If criticalProcess AndAlso Not myForce Then
        message = String.Format(CultureInfo.CurrentCulture, _
            "The process ""{0}"" is a critical process and " & _
            "should not be stopped. " & _
            "Are you sure you wish to stop the process?",

processName)

        ' It is possible that ProcessRecord is called multiple
        ' when objects are received as inputs from a pipeline.
        ' So, to retain YesToAll and NoToAll input that the
        ' user may enter across multiple calls to this
        ' function, they are stored as private members of the
        ' Cmdlet.
        If Not ShouldContinue(message, "Warning!", yesToAll,
noToAll) Then

```

```

        Return
    End If
End If

' Display a warning information if stopping a critical
' process
If criticalProcess Then
    message = String.Format(CultureInfo.CurrentCulture, _
        "Stopping the critical process ""{0}"".", processName)
    WriteWarning(message)
End If

Try
    ' Stop the process.
    process.Kill()
Catch e As Exception
    If TypeOf e Is Win32Exception OrElse TypeOf e Is
SystemException _
        OrElse TypeOf e Is InvalidOperationException Then
        ' This process could not be stopped so write
        ' a non-terminating error.
        WriteError(New ErrorRecord(e, _
            "CouldNotStopProcess", ErrorCategory.CloseError,
process))
        Return
    Else
        Throw
    End If
End Try

' Write a user-level message to the pipeline. These are
' intended to give the user detailed information on the
' operations performed by the Cmdlet. These messages will
' appear with the -Verbose option.
message = String.Format(CultureInfo.CurrentCulture, _
    "Stopped process ""{0}"""", pid {1}.", processName,
process.Id)

WriteVerbose(message)

' If the -PassThru command line argument is
' specified, pass the terminated process on.
If myPassThru Then
    ' Write a debug message to the host which will be helpful
    ' in troubleshooting a problem. All debug messages
    ' will appear with the -Debug option
    message = String.Format(CultureInfo.CurrentCulture, _
        "Writing process ""{0}"" to pipeline", processName)
    WriteDebug(message)
    WriteObject(process)
End If

End Sub 'SafeStopProcess

```

```

''' <summary>
''' Stop processes based on their names (using the
''' ParameterSetName as ProcessName)
''' </summary>
Private Sub ProcessByName()
    Dim allProcesses As ArrayList = Nothing

    ' Get a list of all processes.
    Try
        allProcesses = New ArrayList(Process.GetProcesses())
    Catch ioe As InvalidOperationException
        MyBase.ThrowTerminatingError(New ErrorRecord(ioe, _
            "UnableToAccessProcessList", _
            ErrorCategory.InvalidOperation, Nothing))
    End Try

    ' If a name parameter is passed to cmdlet, get
    ' the associated process(es).
    ' Write a non-terminating error for failure to
    ' retrieve a process
    Dim name As String
    For Each name In processNames
        ' The allProcesses array list is passed as a reference
because
        ' any process whose name cannot be obtained will be removed
        ' from the list so that its not compared the next time.
        Dim processes As ArrayList = SafeGetProcessesByName(name, _
            allProcesses)

        ' If no processes were found write a non-terminating error.
        If processes.Count = 0 Then
            WriteError(New ErrorRecord( _
                New Exception("Process not found."), _
                "ProcessNotFound", ErrorCategory.ObjectNotFound,
name))
        Else
            ' Otherwise terminate all processes in the list.
            Dim process As Process
            For Each process In processes
                SafeStopProcess(process)
            Next process
        End If

        Next name

    End Sub 'ProcessByName

''' <summary>
''' Stop processes based on their ids (using the
''' ParameterSetName as ProcessIds)
''' </summary>
Friend Sub ProcessById()
    Dim processId As Integer
    For Each processId In processIds
        Dim process As Process = Nothing

```

```

        Try
            process =
System.Diagnostics.Process.GetProcessById(processId)

                ' Write a debug message to the host which will be
helpful
                ' in troubleshooting a problem. All debug messages
                ' will appear with the -Debug option
                Dim message As String = String.Format( _
                    CultureInfo.CurrentCulture, _
                    "Acquired process for pid : {0}", process.Id)
                WriteDebug(message)
            Catch ae As ArgumentException
                Dim message As String = String.Format( _
                    CultureInfo.CurrentCulture, _
                    "The process id {0} could not be found", processId)
                WriteVerbose(message)
                WriteError(New ErrorRecord(ae, _
                    "ProcessIdNotFound", _
                    ErrorCategory.ObjectNotFound, processId))
                GoTo ContinueForEach1
            End Try

                SafeStopProcess(process)
ContinueForEach1:
    Next processId

End Sub 'ProcessById ' ProcessById
#End Region

#Region "Private Data"

    Private yesToAll, noToAll As Boolean

    ''' <summary>
    ''' Partial list of critical processes that should not be
    ''' stopped. Lower case is used for case insensitive matching.
    ''' </summary>
    Private criticalProcessNames As New ArrayList( _
        New String() {"system", "winlogon", "spoolsv", "calc"})

#End Region

End Class 'StopProcCommand

#End Region

#Region "PowerShell snap-in"
    ''' <summary>
    ''' Create this sample as a PowerShell snap-in
    ''' </summary>
    <RunInstaller(True)> _
    Public Class StopProcPSSnapIn04
        Inherits PSSnapIn

```

```

''' <summary>
''' Create an instance of the StopProcPSSnapIn04
''' </summary>
Public Sub New()

End Sub 'New

''' <summary>
''' Get a name for this PowerShell snap-in. This name will
''' be used in registering this PowerShell snap-in.
''' </summary>
Public Overrides ReadOnly Property Name() As String
    Get
        Return "StopProcPSSnapIn04"
    End Get
End Property

''' <summary>
''' Vendor information for this PowerShell snap-in.
''' </summary>

Public Overrides ReadOnly Property Vendor() As String
    Get
        Return "Microsoft"
    End Get
End Property

''' <summary>
''' Gets resource information for vendor. This is a string of
format:
''' resourceBaseName,resourceName.
''' </summary>
Public Overrides ReadOnly Property VendorResource() As String
    Get
        Return "StopProcPSSnapIn04,Microsoft"
    End Get
End Property

''' <summary>
''' Description of this PowerShell snap-in.
''' </summary>
Public Overrides ReadOnly Property Description() As String
    Get
        Return "This is a PowerShell snap-in that includes " & _
               "the Stop-Proc cmdlet."
    End Get
End Property
End Class 'StopProcPSSnapIn04

#End Region

End Namespace

```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

Runspace01 Code Samples

Article • 09/17/2021

Here are the code samples for the runspace described in [Creating a Console Application That Runs a Specified Command](#). The command that is invoked in the runspace is the `Get-Process` cmdlet.

For complete sample code, see the following topics.

[] Expand table

Language	Topic
C#	Runspace01 (C#) Code Sample
VB.NET	Runspace01 (VB.NET) Code Sample

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Runspace01 (C#) Code Sample

Article • 09/17/2021

Here are the code samples for the runspace described in [Creating a Console Application That Runs a Specified Command](#). To do this, the application invokes a runspace, and then invokes a command. (Note that this application does not specify runspace configuration information, nor does it explicitly create a pipeline). The command that is invoked is the `Get-Process` cmdlet.

ⓘ Note

You can download the C# source file (`runspace01.cs`) for this runspace using the Microsoft Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the `<PowerShell Samples>` directory.

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Management.Automation;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace01
    {
        /// <summary>
        /// This sample uses the PowerShell class to execute
        /// the get-process cmdlet synchronously. The name and
        /// handlecount are then extracted from the PSObjects
        /// returned and displayed.
        /// </summary>
        /// <param name="args">Parameter not used.</param>
        /// <remarks>
        /// This sample demonstrates the following:
        /// 1. Creating a PowerShell object to run a command.
        /// 2. Adding a command to the pipeline of the PowerShell object.
        /// 3. Running the command synchronously.
        /// 4. Using PSObject objects to extract properties from the objects
    }
}
```

```
///      returned by the command.  
/// </remarks>  
private static void Main(string[] args)  
{  
    // Create a PowerShell object. Creating this object takes care of  
    // building all of the other data structures needed to run the  
    // command.  
    using (PowerShell powershell = PowerShell.Create().AddCommand("get-  
process"))  
    {  
        Console.WriteLine("Process           HandleCount");  
        Console.WriteLine("-----");  
  
        // Invoke the command synchronously and display the  
        // ProcessName and HandleCount properties of the  
        // objects that are returned.  
        foreach (PSObject result in powershell.Invoke())  
        {  
            Console.WriteLine(  
                "{0,-20} {1}",  
                result.Members["ProcessName"].Value,  
                result.Members["HandleCount"].Value);  
        }  
    }  
  
    System.Console.WriteLine("Hit any key to exit...");  
    System.Console.ReadKey();  
}  
}  
}
```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review
issues and pull requests. For
more information, see [our](#)
contributor guide.



PowerShell feedback

PowerShell is an open source
project. Select a link to provide
feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Runspace01 (VB.NET) Code Sample

Article • 09/17/2021

Here are the code samples for the runspace described in [Creating a Console Application That Runs a Specified Command](#). To do this, the application invokes a runspace, and then invokes a command. (Note that this application does not specify runspace configuration information, nor does it explicitly create a pipeline.) The command that is invoked is the `Get-Process` cmdlet.

Code Sample

```
VB

Imports System
Imports System.Collections.Generic
Imports System.Text
Imports System.Management.Automation
Imports System.Management.Automation.Host
Imports System.Management.Automation.Runspaces

Namespace Microsoft.Samples.PowerShell.Runspaces

Module Runspace01
    ' <summary>
    ' This sample uses the RunspaceInvoke class to execute
    ' the Get-Process cmdlet synchronously. The name and
    ' handlecount are then extracted from the PSObjects
    ' returned and displayed.
    ' </summary>
    ' <param name="args">Unused</param>
    ' <remarks>
    ' This sample demonstrates the following:
    ' 1. Creating an instance of the RunspaceInvoke class.
    ' 2. Using this instance to invoke a PowerShell command.
    ' 3. Using PSObject to extract properties from the objects
    '    returned by this command.
    ' </remarks>
    Sub Main()
        ' Create an instance of the RunspaceInvoke class.
        ' This takes care of all building all of the other
        ' data structures needed...
        Dim invoker As RunspaceInvoke = New RunspaceInvoke()

        Console.WriteLine("Process           HandleCount")
        Console.WriteLine("-----")

        ' Now invoke the runspace and display the objects that are
        ' returned...
        For Each result As PSObject In invoker.Invoke("Get-Process")
```

```
Console.WriteLine("{0,-20} {1}", _  
    result.Members("ProcessName").Value, _  
    result.Members("HandleCount").Value)  
    Next  
    System.Console.WriteLine("Hit any key to exit...")  
    System.Console.ReadKey()  
End Sub  
End Module  
End Namespace
```

See Also

[Windows PowerShell SDK](#)

Runspace02 Code Samples

Article • 09/17/2021

Here is the source code for the Runspace02 sample. This sample uses the [System.Management.Automation.RunspacesInvoke](#) class to execute the `Get-Process` cmdlet synchronously. Windows Forms and data binding are then used to display the results in a DataGridView control.

For complete sample code, see the following topics.

[+] [Expand table](#)

Language	Topic
C#	Runspace02 (C#) Code Sample
VB.NET	Runspace02 (VB.NET) Code Sample

See Also

[Windows PowerShell SDK](#)

Runspace02 (C#) Code Sample

Article • 09/17/2021

Here is the C# source code for the Runspace02 sample. This sample uses the [System.Management.Automation.RunspacesInvoke](#) class to execute the `Get-Process` cmdlet synchronously. Windows Forms and data binding are then used to display the results in a DataGridView control

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using System.Windows.Forms;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace02
    {
        /// <summary>
        /// This method creates the form where the output is displayed.
        /// </summary>
        private static void CreateForm()
        {
            Form form = new Form();
            DataGridView grid = new DataGridView();
            form.Controls.Add(grid);
            grid.Dock = DockStyle.Fill;

            // Create a PowerShell object. Creating this object takes care of
            // building all of the other data structures needed to run the
            command.

            using (PowerShell powershell = PowerShell.Create())
            {
                powershell.AddCommand("get-process").AddCommand("sort-
object").AddArgument("ID");
                if (Runspace.DefaultRunspace == null)
                {
                    Runspace.DefaultRunspace = powershell.Runspace;
                }
            }
        }
    }
}
```

```
Collection<PSObject> results = powershell.Invoke();

// The generic collection needs to be re-wrapped in an ArrayList
// for data-binding to work.
ArrayList objects = new ArrayList();
objects.AddRange(results);

// The DataGridView will use the PSObjectTypeDescriptor type
// to retrieve the properties.
grid.DataSource = objects;
}

form.ShowDialog();
}

/// <summary>
/// This sample uses a PowerShell object to run the
/// Get-Process cmdlet synchronously. Windows Forms and
/// data binding are then used to display the results in a
/// DataGridView control.
/// </summary>
/// <param name="args">Parameter not used.</param>
/// <remarks>
/// This sample demonstrates the following:
/// 1. Creating a PowerShell object.
/// 2. Adding commands and arguments to the pipeline of
///    the powershell object.
/// 3. Running the commands synchronously.
/// 4. Using a DataGridView control to display the output
///    of the PowerShell object in a Windows Forms application.
/// </remarks>
private static void Main(string[] args)
{
    Runspace02.CreateForm();
}
}
```

See Also

[Windows PowerShell SDK](#)

Runspace02 (VB.NET) Code Sample

Article • 09/17/2021

Here is the VB.NET source code for the Runspace02 sample. This sample uses the [System.Management.Automation.RunspacesInvoke](#) class to execute the `Get-Process` cmdlet synchronously. Windows Forms and data binding are then used to display the results in a DataGridView control.

Code Sample

VB

```
Imports System
Imports System.Collections
Imports System.Collections.ObjectModel
Imports System.Windows.Forms
Imports System.Management.Automation.Runspaces
Imports System.Management.Automation

Namespace Microsoft.Samples.PowerShell.Runspaces

    Class Runspace02

        Shared Sub CreateForm()
            Dim form As New Form()
            Dim grid As New DataGridView()
            form.Controls.Add(grid)
            grid.Dock = DockStyle.Fill

            ' Create an instance of the RunspaceInvoke class.
            ' This takes care of all building all of the other
            ' data structures needed...
            Dim invoker As New RunspaceInvoke()

            Dim results As Collection(Of PSObject) = _
                invoker.Invoke("Get-Process | Sort-Object ID")

            ' The generic collection needs to be re-wrapped in an ArrayList
            ' for data-binding to work...
            Dim objects As New ArrayList()
            objects.AddRange(results)

            ' The DataGridView will use the PSObjectTypeDescriptor type
            ' to retrieve the properties.
            grid.DataSource = objects

            form.ShowDialog()

        End Sub 'CreateForm
    End Class
End Namespace
```

```
''' <summary>
''' This sample uses the RunspaceInvoke class to execute
''' the Get-Process cmdlet synchronously. Windows Forms and data
''' binding are then used to display the results in a
''' DataGridView control.
''' </summary>
''' <param name="args">Unused</param>
''' <remarks>
''' This sample demonstrates the following:
''' 1. Creating an instance of the RunspaceInvoke class.
''' 2. Using this instance to invoke a PowerShell command.
''' 3. Using the output of RunspaceInvoke in a DataGridView
'''     in a Windows Forms application
''' </remarks>
Shared Sub Main(ByVal args() As String)
    Runspace02.CreateForm()
End Sub 'Main

End Class 'Runspace02

End Namespace
```

See Also

[Windows PowerShell SDK](#)

RunSpace03 Code Samples

Article • 09/17/2021

Here are the code samples for the runspace described in "Creating a Console Application That Runs a Specified Script".

Note

You can download the C# source file (runspace03.cs) and the VB.NET source file (runspace03.vb) for this sample using the Microsoft Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory.

For complete sample code, see the following topics.

 Expand table

Language	Topic
C#	RunSpace03 (C#) Code Sample
VB.NET	RunSpace03 (VB.NET) Code Sample

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

RunSpace03 (C#) Code Sample

Article • 03/24/2025

Here is the C# source code for the console application described in "Creating a Console Application That Runs a Specified Script". This sample uses the [System.Management.Automation.RunspaceInvoke](#) class to execute a script that retrieves process information by using the list of process names passed into the script. It shows how to pass input objects to a script and how to retrieve error objects as well as the output objects.

ⓘ Note

You can download the C# source file (runspace03.cs) for this sample using the Microsoft Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory.

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace03
    {
        /// <summary>
        /// This sample uses the PowerShell class to execute
        /// a script that retrieves process information for the
        /// list of process names passed into the script.
        /// It shows how to pass input objects to a script and
        /// how to retrieve error objects as well as the output objects.
        /// </summary>
        /// <param name="args">Parameter not used.</param>
        /// <remarks>
        /// This sample demonstrates the following:
    }
}
```

```

    /// 1. Creating an instance of the PowerShell class.
    /// 2. Using this instance to execute a string as a PowerShell
script.
    /// 3. Passing input objects to the script from the calling program.
    /// 4. Using PSObject to extract and display properties from the
objects
    ///     returned by this command.
    /// 5. Retrieving and displaying error records that were generated
    ///     during the execution of that script.
    /// </remarks>
private static void Main(string[] args)
{
    // Define a list of processes to look for
    string[] processNames = new string[]
    {
        "lsass", "nosuchprocess", "services", "nosuchprocess2"
    };

    // The script to run to get these processes. Input passed
    // to the script will be available in the $input variable.
    string script = "$input | get-process -name ${_}";

    // Create an instance of the PowerShell class.
    using (PowerShell powershell = PowerShell.Create())
    {
        powershell.AddScript(script);

        Console.WriteLine("Process           HandleCount");
        Console.WriteLine("-----");

        // Now invoke the PowerShell and display the objects that
are
        // returned...
        foreach (PSObject result in powershell.Invoke(processNames))
        {
            Console.WriteLine(
                "{0,-20} {1}",
                result.Members["ProcessName"].Value,
                result.Members["HandleCount"].Value);
        }

        // Now process any error records that were generated while
running the script.
        Console.WriteLine("\nThe following non-terminating errors
occurred:\n");
        PSDataCollection<ErrorRecord> errors =
powershell.Streams.Error;
        if (errors != null && errors.Count > 0)
        {
            foreach (ErrorRecord err in errors)
            {
                System.Console.WriteLine("    error: {0}",
err.ToString());
            }
        }
    }
}

```

```
        }

        System.Console.WriteLine("\nHit any key to exit...");
        System.Console.ReadKey();
    }
}
```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

RunSpace03 (VB.NET) Code Sample

Article • 09/17/2021

Here is the VB.NET source code for the console application described in "Creating a Console Application That Runs a Specified Script". This sample uses the [System.Management.Automation.RunspaceInvoke](#) class to execute a script that retrieves process information for the list of process names passed into the script. It shows how to pass input objects to a script and how to retrieve error objects as well as the output objects.

ⓘ Note

You can download the VB.NET source file (runspace03.vb) for this sample by using the Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory.

Code Sample

VB

```
Imports System
Imports System.Collections
Imports System.Collections.Generic
Imports System.Collections.ObjectModel
Imports System.Text
Imports Microsoft.VisualBasic
Imports System.Management.Automation
Imports System.Management.Automation.Host
Imports System.Management.Automation.Runspaces

Namespace Microsoft.Samples.PowerShell.Runspaces

    Class Runspace03

        ''' <summary>
        ''' This sample uses the RunspaceInvoke class to execute
        ''' a script that retrieves process information for the
        ''' list of process names passed into the script.
        ''' It shows how to pass input objects to a script and
        ''' how to retrieve error objects as well as the output objects.
        ''' </summary>
        ''' <param name="args">Unused</param>
        ''' <remarks>
```

```

''' This sample demonstrates the following:
''' 1. Creating an instance of the RunspaceInvoke class.
''' 2. Using this instance to execute a string as a PowerShell
script.
''' 3. Passing input objects to the script from the calling program.
''' 4. Using PSObject to extract and display properties from the
objects
'''      returned by this command.
''' 5. Retrieving and displaying error records that were generated
'''      during the execution of that script.
''' </remarks>
Shared Sub Main(ByVal args() As String)
    ' Define a list of processes to look for
    Dim processNames() As String = {"lsass", "nosuchprocess", _
        "services", "nosuchprocess2"}

    ' The script to run to get these processes. Input passed
    ' to the script will be available in the $input variable.
    Dim script As String = "$input | Get-Process -Name {$_}"

    ' Create an instance of the RunspaceInvoke class.
    Dim invoker As New RunspaceInvoke()

    Console.WriteLine("Process                  HandleCount")
    Console.WriteLine("-----")

    ' Now invoke the runspace and display the objects that are
    ' returned...
    Dim errors As System.Collections.IList = Nothing
    Dim result As PSObject
    For Each result In invoker.Invoke(script, processNames, errors)
        Console.WriteLine("{0,-20} {1}", _
            result.Members("ProcessName").Value, _
            result.Members("HandleCount").Value)
    Next result

    ' Now process any error records that were generated while
    ' running the script.
    Console.WriteLine(vbCrLf & _
        "The following non-terminating errors occurred:" & vbCrLf)
    If Not (errors Is Nothing) AndAlso errors.Count > 0 Then
        Dim err As PSObject
        For Each err In errors
            System.Console.WriteLine("    error: {0}", _
                err.ToString())
        Next err
    End If
    System.Console.WriteLine(vbCRLF & "Hit any key to exit...")
    System.Console.ReadKey()

End Sub 'Main

End Class 'Runspace03

End Namespace

```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

RunSpace04 Code Samples

Article • 09/17/2021

Here is a code sample for a runspace that uses the [System.Management.Automation.RunspaceInvoke](#) class to execute a script that generates a terminating error. The host application is responsible for catching the error and interpreting the error record.

Note

You can download the VB.NET source file (Runspace04.vb) for this runspace using the Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory.

For complete sample code, see the following topics.

 Expand table

Language	Topic
VB.NET	Runspace04 (VB.NET) Code Sample

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

RunSpace04 (VB.NET) Code Sample

Article • 09/17/2021

Here is the VB.NET source code for the Runspace04 sample. This sample uses the [System.Management.Automation.RunspaceInvoke](#) class to execute a script that generates a terminating error. The host application is responsible for catching the error and interpreting the error record.

ⓘ Note

You can download the VB.NET source file (runspace02.vb) for this sample by using the Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#).

The downloaded source files are available in the <PowerShell Samples> directory.

Code Sample

VB

```
Imports System
Imports System.Collections
Imports System.Collections.Generic
Imports System.Collections.ObjectModel
Imports System.Text
Imports Microsoft.VisualBasic
Imports System.Management.Automation
Imports System.Management.Automation.Host
Imports System.Management.Automation.Runspaces

Namespace Microsoft.Samples.PowerShell.Runspaces

    Class Runspace04

        ''' <summary>
        ''' This sample uses the RunspaceInvoke class to execute
        ''' a script. This script will generate a terminating
        ''' exception that the caller should catch and process.
        ''' </summary>
        ''' <param name="args">Unused</param>
        ''' <remarks>
        ''' This sample demonstrates the following:
        ''' 1. Creating an instance of the RunspaceInvoke class.
        ''' 2. Using this instance to execute a string as a PowerShell
        ''' script.
    End Class
End Namespace
```

```

    ''' 3. Passing input objects to the script from the calling program.
    ''' 4. Using PSObject to extract and display properties from the
objects
    '''      returned by this command.
    ''' 5. Retrieving and displaying error records that may be generated
    '''      during the execution of that script.
    ''' 6. Catching and displaying terminating exceptions generated
    '''      by the script being run.
    ''' </remarks>
Shared Sub Main(ByVal args() As String)
    ' Define a list of patterns to use in matching
    ' Note that the fourth pattern is not a valid regular
    ' expression so it will cause a terminating exception to
    ' be thrown when used in Select-String.
    Dim patterns() As String = {"aa", "bc", "ab*c", "*", "abc"}

    ' The script to run to use the patterns. Input passed
    ' to the script will be available in the $input variable.
    Dim script As String = "$input | where {" & _
        " Select-String $_ -InputObject 'abc' }"

    ' Create an instance of the RunspaceInvoke class.
    Dim invoker As New RunspaceInvoke()

    ' Invoke the runspace. Because of the bad regular expression,
    ' no objects will be returned. Instead, an exception will be
    ' thrown.
    Try
        Dim errors As System.Collections.IList = Nothing
        Dim result As PSObject
        For Each result In invoker.Invoke(script, patterns, errors)
            Console.WriteLine("'{0}'", result.ToString())
        Next result

        ' Now process any error records that were generated
        ' while running the script.
        Console.WriteLine(vbCrLf & _
            "The following non-terminating errors occurred:" &
vbCrLf)
        If Not (errors Is Nothing) AndAlso errors.Count > 0 Then
            Dim err As PSObject
            For Each err In errors
                System.Console.WriteLine("    error: {0}",
err.ToString())
            Next err
        End If
        Catch runtimeException As RuntimeException
            ' Trap any exception generated by the script. These
exceptions
                ' will all be derived from RuntimeException.
                System.Console.WriteLine("Runtime exception: {0}: {1}" & _
                    vbCrLf + "{2}", _
runtimeException.ErrorRecord.InvocationInfo.InvocationName, _
                    runtimeException.Message, _

```

```
runtimeException.ErrorRecord.InvocationInfo.PositionMessage)
    End Try

    System.Console.WriteLine(vbCrLf + "Hit any key to exit...")
    System.Console.ReadKey()

End Sub 'Main
End Class 'Runspace04

End Namespace
```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

RunSpace05 Code Sample

Article • 09/17/2021

Here is the source code for the Runspace05 sample that is described in [Configuring a Runspace Using RunspaceConfiguration](#). This sample shows how to create the runspace configuration information, create a runspace, create a pipeline with a single command, and then execute the pipeline. The command that is executed is the `Get-Process` cmdlet.

ⓘ Note

You can download the C# source file (`runspace05.cs`) by using the Microsoft Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the `<PowerShell Samples>` directory.

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace05
    {
        /// <summary>
        /// This sample uses an initial session state to create a runspace. The
        /// sample
        /// invokes a command from a PowerShell snap-in present in the console
        /// file.
        /// </summary>
        /// <param name="args">Parameter not used.</param>
        /// <remarks>
        /// This sample assumes that user has the GetProcessSample01.dll that is
        /// produced
        /// by the GetProcessSample01 sample copied to the current directory.
    }
}
```

```

/// This sample demonstrates the following:
/// 1. Creating a default initial session state.
/// 2. Creating a runspace using the default initial session state.
/// 3. Creating a PowerShell object that uses the runspace.
/// 4. Adding the get-proc cmdlet to the PowerShell object from a
///    snap-in.
/// 5. Using PSObject objects to extract and display properties from
///    the objects returned by the cmdlet.
/// </remarks>
private static void Main(string[] args)
{
    // Create the default initial session state. The default initial
    // session state contains all the elements provided by Windows
PowerShell.
    InitialSessionState iss = InitialSessionState.CreateDefault();
    PSSnapInException warning;
    iss.ImportPSSnapIn("GetProcPSSnapIn01", out warning);

    // Create a runspace. Notice that no PSHost object is supplied to the
    // CreateRunspace method so the default host is used. See the Host
    // samples for more information on creating your own custom host.
    using (Runspace myRunSpace = RunspaceFactory.CreateRunspace(iss))
    {
        myRunSpace.Open();

        // Create a PowerShell object.
        using (PowerShell powershell = PowerShell.Create())
        {
            // Add the Cmdlet and specify the runspace.
            powershell.AddCommand("GetProcPSSnapIn01\\get-proc");
            powershell.Runspace = myRunSpace;

            // Run the cmdlet synchronously.
            Collection<PSObject> results = powershell.Invoke();

            Console.WriteLine("Process                  HandleCount");
            Console.WriteLine("-----");

            // Display the results.
            foreach (PSObject result in results)
            {
                Console.WriteLine(
                    "{0,-20} {1}",
                    result.Members["ProcessName"].Value,
                    result.Members["HandleCount"].Value);
            }
        }

        // Close the runspace to release any resources.
        myRunSpace.Close();
    }
    System.Console.WriteLine("Hit any key to exit...");
    System.Console.ReadKey();
}

```

```
}
```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

RunSpace06 Code Sample

Article • 09/17/2021

Here is the source code for the Runspace06 sample described in [Configuring a Runspace Using a Windows PowerShell Snap-in](#). This sample application creates a runspace based on a Windows PowerShell snap-in, which is then used to run a pipeline with a single command. To do this, the application creates the runspace configuration information, creates a runspace, creates a pipeline with a single command, and then executes the pipeline.

ⓘ Note

You can download the C# source file (runspace06.cs) by using the Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory.

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace06
    {
        /// <summary>
        /// This sample uses an initial session state to create a runspace.
        /// The sample invokes a command from binary module that is loaded by
        /// the
        /// initial session state.
        /// </summary>
        /// <param name="args">Parameter not used.</param>
        /// <remarks>
        /// This sample assumes that user has the GetProcessSample02.dll that is
        /// produced by the GetProcessSample02 sample copied to the current
```

```
directory.

/// This sample demonstrates the following:
/// 1. Creating a default initial session state.
/// 2. Creating a runspace using the initial session state.
/// 3. Creating a PowerShell object that uses the runspace.
/// 4. Adding the get-proc cmdlet to the PowerShell object from a
///    module.
/// 5. Using PSObject objects to extract and display properties from
///    the objects returned by the cmdlet.
/// </remarks>
private static void Main(string[] args)
{
    // Create an initial session state.
    InitialSessionState iss = InitialSessionState.CreateDefault();
    iss.ImportPSModule(new string[] { @"\.\GetProcessSample02.dll" });

    // Create a runspace. Notice that no PSHost object is supplied to the
    // CreateRunspace method so the default host is used. See the Host
    // samples for more information on creating your own custom host.
    using (Runspace myRunSpace = RunspaceFactory.CreateRunspace(iss))
    {
        myRunSpace.Open();

        // Create a PowerShell object.
        using (PowerShell powershell = PowerShell.Create())
        {
            // Add the cmdlet and specify the runspace.
            powershell.AddCommand(@"GetProcessSample02\get-proc");
            powershell.Runspace = myRunSpace;

            Collection<PSObject> results = powershell.Invoke();

            Console.WriteLine("Process           HandleCount");
            Console.WriteLine("-----");

            // Display the results.
            foreach (PSObject result in results)
            {
                Console.WriteLine(
                    "{0,-20} {1}",
                    result.Members["ProcessName"].Value,
                    result.Members["HandleCount"].Value);
            }
        }

        // Close the runspace to release any resources.
        myRunSpace.Close();
    }

    System.Console.WriteLine("Hit any key to exit...");
    System.Console.ReadKey();
}
```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

RunSpace07 Code Sample

Article • 09/17/2021

Here is the source code for the Runspace07 sample described in [Creating a Console Application That Adds Commands to a Pipeline](#). This sample application creates a runspace, creates a pipeline, adds two commands to the pipeline, and then executes the pipeline. The commands added to the pipeline are the `Get-Process` and `Measure-Object` cmdlets.

ⓘ Note

You can download the C# source file (`runspace07.cs`) using the Microsoft Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the `<PowerShell Samples>` directory.

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace07
    {
        /// <summary>
        /// This sample shows how to create a runspace and how to run
        /// commands using a PowerShell object. It builds a pipeline
        /// that runs the get-process cmdlet, which is piped to the measure-
        object
        /// cmdlet to count the number of processes running on the system.
        /// </summary>
        /// <param name="args">Parameter is not used.</param>
        /// <remarks>
        /// This sample demonstrates the following:
        /// 1. Creating a runspace using the RunspaceFactory class.
    }
}
```

```
/// 2. Creating a PowerShell object
/// 3. Adding individual cmdlets to the PowerShell object.
/// 4. Running the cmdlets synchronously.
/// 5. Working with PSObject objects to extract properties
///     from the objects returned by the cmdlets.
/// </remarks>
private static void Main(string[] args)
{
    Collection<PSObject> result;      // Will hold the result
                                       // of running the cmdlets.

    // Create a runspace. We can not use the RunspaceInvoke class
    // because we need to get at the underlying runspace to
    // explicitly add the commands. Notice that no PSHost object is
    // supplied to the CreateRunspace method so the default host is
    // used. See the Host samples for more information on creating
    // your own custom host.
    using (Runspace myRunSpace = RunspaceFactory.CreateRunspace())
    {
        myRunSpace.Open();

        // Create a PowerShell object and specify the runspace.
        PowerShell powershell = PowerShell.Create();
        powershell.Runspace = myRunSpace;

        // Use the using statement so we dispose of the PowerShell object
        // when we're done.
        using (powershell)
        {
            // Add the get-process cmdlet to the PowerShell object. Notice
            // we are specify the name of the cmdlet, not a script.
            powershell.AddCommand("get-process");

            // Add the measure-object cmdlet to count the number
            // of objects being returned. Commands are always added to the end
            // of the pipeline.
            powershell.AddCommand("measure-object");

            // Run the cmdlets synchronously and save the objects returned.
            result = powershell.Invoke();
        }

        // Even after disposing of the pipeline, we still need to set
        // the powershell variable to null so that the garbage collector
        // can clean it up.
        powershell = null;

        // Display the results of running the commands (checking that
        // everything is ok first).
        if (result == null || result.Count != 1)
        {
            throw new InvalidOperationException(
                "pipeline.Invoke() returned the wrong number of
objects");
        }
    }
}
```

```
PSMemberInfo count = result[0].Properties["Count"];
if (count == null)
{
    throw new InvalidOperationException(
        "The object returned doesn't have a 'count' property");
}

Console.WriteLine(
    "Runspace07: The get-process cmdlet returned {0}
objects",
    count.Value);

// Close the runspace to release any resources.
myRunSpace.Close();
}

System.Console.WriteLine("Hit any key to exit...");
System.Console.ReadKey();
}
}
}
```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

RunSpace08 Code Sample

Article • 09/17/2021

Here is the source code for the Runspace08 sample described in [Creating a Console Application That Adds Parameters to a Command](#). This sample application creates a runspace, creates a pipeline, adds two commands to the pipeline, adds two parameters to the second command, and then executes the pipeline. The commands that are added to the pipeline are the `Get-Process` and `Sort-Object` cmdlets.

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace08
    {
        /// <summary>
        /// This sample shows how to use a PowerShell object to run commands.
        The
        /// PowerShell object builds a pipeline that include the get-process
        cmdlet,
        /// which is then piped to the sort-object cmdlet. Parameters are added
        to the
        /// sort-object cmdlet to sort the HandleCount property in descending
        order.
        /// </summary>
        /// <param name="args">Parameter is not used.</param>
        /// <remarks>
        /// This sample demonstrates:
        /// 1. Creating a PowerShell object
        /// 2. Adding individual commands to the PowerShell object.
        /// 3. Adding parameters to the commands.
        /// 4. Running the pipeline of the PowerShell object synchronously.
        /// 5. Working with PSObject objects to extract properties
        ///     from the objects returned by the commands.
        /// </remarks>
        private static void Main(string[] args)
```

```
{  
    Collection<PSObject> results; // Holds the result of the pipeline  
    execution.  
  
    // Create the PowerShell object. Notice that no runspace is specified  
    so a  
    // new default runspace is used.  
    PowerShell powershell = PowerShell.Create();  
  
    // Use the using statement so that we can dispose of the PowerShell  
    object  
    // when we are done.  
    using (powershell)  
    {  
        // Add the get-process cmdlet to the pipeline of the PowerShell  
        object.  
        powershell.AddCommand("get-process");  
  
        // Add the sort-object cmdlet and its parameters to the pipeline of  
        // the PowerShell object so that we can sort the HandleCount  
        property  
        // in descending order.  
        powershell.AddCommand("sort-  
        object").AddParameter("descending").AddParameter("property", "handlecount");  
  
        // Run the pipeline of the PowerShell object synchronously.  
        results = powershell.Invoke();  
    }  
  
    // Even after disposing of the PowerShell object, we still  
    // need to set the powershell variable to null so that the  
    // garbage collector can clean it up.  
    powershell = null;  
  
    Console.WriteLine("Process           HandleCount");  
    Console.WriteLine("-----");  
  
    // Display the results returned by the commands.  
    foreach (PSObject result in results)  
    {  
        Console.WriteLine(  
            "{0,-20} {1}",  
            result.Members["ProcessName"].Value,  
            result.Members["HandleCount"].Value);  
    }  
  
    System.Console.WriteLine("Hit any key to exit...");  
    System.Console.ReadKey();  
}  
}
```

See Also

Windows PowerShell SDK

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

RunSpace09 Code Sample

Article • 09/17/2021

This sample application creates and opens a runspace, creates and asynchronously invokes a pipeline, and then uses pipeline events to process the script asynchronously. The script that is run by this application creates the integers 1 through 10 in 0.5-second intervals (500 ms).

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Diagnostics;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    /// <summary>
    /// This class contains the Main entry point for this host application.
    /// </summary>
    internal class Runspace09
    {
        /// <summary>
        /// This sample shows how to use a PowerShell object to run a
        /// script that generates the numbers from 1 to 10 with delays
        /// between each number. The pipeline of the PowerShell object
        /// is run asynchronously and events are used to handle the output.
        /// </summary>
        /// <param name="args">This parameter is not used.</param>
        /// <remarks>
        /// This sample demonstrates the following:
        /// 1. Creating a PowerShell object.
        /// 2. Adding a script to the pipeline of the PowerShell object.
        /// 3. Using the BeginInvoke method to run the pipeline asynchronously.
        /// 4. Using the events of the PowerShell object to process the
        ///     output of the script.
        /// 5. Using the PowerShell.Stop() method to interrupt an executing
        pipeline.
        /// </remarks>
        private static void Main(string[] args)
        {
            Console.WriteLine("Print the numbers from 1 to 10. Hit any key to halt
processing\n");
        }
    }
}
```

```

        using (PowerShell powershell = PowerShell.Create())
    {
        // Add a script to the PowerShell object. The script generates the
        // numbers from 1 to 10 in half second intervals.
        powershell.AddScript("1..10 | foreach {$_ ; start-sleep -milli
500}");

        // Add the event handlers. If we did not care about hooking the
DataAdded
        // event, we would let BeginInvoke create the output stream for us.
        PSDataCollection<PSObject> output = new PSDataCollection<PSObject>
();
        output.DataAdded += new EventHandler<DataAddedEventArgs>
(Output_DataAdded);
        powershell.InvocationStateChanged += new
EventHandler<PSInvocationStateChangedEventArgs>
(Powershell_InvocationStateChanged);

        // Invoke the pipeline asynchronously.
        IAsyncResult asyncResult = powershell.BeginInvoke<PSObject,
PSObject>(null, output);

        // Wait for things to happen. If the user hits a key before the
        // script has completed, then call the PowerShell Stop() method
        // to halt processing.
        Console.ReadKey();
        if (powershell.InvocationStateInfo.State !=
PSInvocationState.Completed)
        {
            // Stop the execution of the pipeline.
            Console.WriteLine("\nStopping the pipeline!\n");
            powershell.Stop();

            // Wait for the Windows PowerShell state change messages to be
displayed.
            System.Threading.Thread.Sleep(500);
            Console.WriteLine("\nPress a key to exit");
            Console.ReadKey();
        }
    }

    /// <summary>
    /// The output data added event handler. This event is called when
    /// data is added to the output pipe. It reads the data that is
    /// available and displays it on the console.
    /// </summary>
    /// <param name="sender">The output pipe this event is associated with.
</param>
    /// <param name="e">Parameter is not used.</param>
    private static void Output_DataAdded(object sender, DataAddedEventArgs
e)
    {
        PSDataCollection<PSObject> myp = (PSDataCollection<PSObject>)sender;

```

```
Collection<PSObject> results = myp.ReadAll();
foreach (PSObject result in results)
{
    Console.WriteLine(result.ToString());
}
}

/// <summary>
/// This event handler is called when the pipeline state is changed.
/// If the state change is to Completed, the handler issues a message
/// asking the user to exit the program.
/// </summary>
/// <param name="sender">This parameter is not used.</param>
/// <param name="e">The PowerShell state information.</param>
private static void Powershell_InvocationStateChanged(object sender,
PSInvocationStateChangedEventArgs e)
{
    Console.WriteLine("PowerShell object state changed: state: {0}\n",
e.InvocationStateInfo.State);
    if (e.InvocationStateInfo.State == PSInvocationState.Completed)
    {
        Console.WriteLine("Processing completed, press a key to exit!");
    }
}
}
```

See Also

[Windows PowerShell SDK](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



PowerShell feedback

PowerShell is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

RunSpace10 Code Sample

Article • 09/17/2021

Here is the source code for the Runspace10 sample. This sample application adds a cmdlet to [System.Management.Automation.Runspaces.RunspaceConfiguration](#) and then uses the modified configuration information to create the runspace.

ⓘ Note

You can download the C# source file (runspace10.cs) by using the Windows Software Development Kit for Windows Vista and Microsoft .NET Framework 3.0 Runtime Components. For download instructions, see [How to Install Windows PowerShell and Download the Windows PowerShell SDK](#). The downloaded source files are available in the <PowerShell Samples> directory.

Code Sample

C#

```
namespace Microsoft.Samples.PowerShell.Runspaces
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Diagnostics;
    using System.Management.Automation;
    using System.Management.Automation.Runspaces;
    using PowerShell = System.Management.Automation.PowerShell;

    #region GetProcCommand

    /// <summary>
    /// Class that implements the GetProcCommand.
    /// </summary>
    [Cmdlet(VerbsCommon.Get, "Proc")]
    public class GetProcCommand : Cmdlet
    {
        #region Cmdlet Overrides

        /// <summary>
        /// For each of the requested process names, retrieve and write
        /// the associated processes.
        /// </summary>
        protected override void ProcessRecord()
        {
            // Get the current processes.
```

```

        Process[] processes = Process.GetProcesses();

        // Write the processes to the pipeline making them available
        // to the next cmdlet. The second argument (true) tells the
        // system to enumerate the array, and send one process object
        // at a time to the pipeline.
        WriteObject(processes, true);
    }

    #endregion Overrides
} // End GetProcCommand class.

#endregion GetProcCommand

/// <summary>
/// This class contains the Main entry point for this host application.
/// </summary>
internal class Runspace10
{
    /// <summary>
    /// This sample shows how to add a cmdlet to an InitialSessionState
    object and then
    /// uses the modified InitialSessionState object when creating a
    Runspace object.
    /// </summary>
    /// <param name="args">Parameter is not used.</param>
    /// This sample demonstrates:
    /// 1. Creating an InitialSessionState object.
    /// 2. Adding a cmdlet to the InitialSessionState object.
    /// 3. Creating a runspace that uses the InitialSessionState object.
    /// 4. Creating a PowerShell object that uses the Runspace object.
    /// 5. Running the pipeline of the PowerShell object synchronously.
    /// 6. Working with PSObject objects to extract properties
    ///     from the objects returned by the pipeline.
    private static void Main(string[] args)
    {
        // Create a default InitialSessionState object. The default
        // InitialSessionState object contains all the elements provided
        // by Windows PowerShell.
        InitialSessionState iss = InitialSessionState.CreateDefault();

        // Add the get-proc cmdlet to the InitialSessionState object.
        SessionStateCmdletEntry ssce = new SessionStateCmdletEntry("get-proc",
typeof(GetProcCommand), null);
        iss.Commands.Add(ssce);

        // Create a Runspace object that uses the InitialSessionState object.
        // Notice that no PSHost object is specified, so the default host is
        used.
        // See the Hosting samples for information on creating your own custom
        host.
        using (Runspace myRunSpace = RunspaceFactory.CreateRunspace(iss))
        {
            myRunSpace.Open();

```

```

    using (PowerShell powershell = PowerShell.Create())
    {
        powershell.Runspace = myRunSpace;

        // Add the get-proc cmdlet to the pipeline of the PowerShell
        // object.
        powershell.AddCommand("get-proc");

        Collection<PSObject> results = powershell.Invoke();

        Console.WriteLine("Process           HandleCount");
        Console.WriteLine("-----");

        // Display the output of the pipeline.
        foreach (PSObject result in results)
        {
            Console.WriteLine(
                "{0,-20} {1}",
                result.Members["ProcessName"].Value,
                result.Members["HandleCount"].Value);
        }
    }

    // Close the runspace to release resources.
    myRunSpace.Close();
}

System.Console.WriteLine("Hit any key to exit...");
System.Console.ReadKey();
}
}
}

```

See Also

[Windows PowerShell Programmer's Guide](#)

[Windows PowerShell SDK](#)

Contributing to PowerShell documentation

Article • 03/30/2025

Thank you for your support of PowerShell!

The Contributor's Guide is a collection of articles that describe the tools and processes we use to create documentation at Microsoft. Some of these guides cover information common to any documentation set published to learn.microsoft.com. Other guides are specific to how we write documentation for PowerShell.

The common articles are available in our centralized [Contributor's Guide](#). The PowerShell-specific guides are available here.

Ways to contribute

There are two ways to contribute. Both contributions are valuable to us.

- [Filing issues](#) helps us identify problems and gaps in our documentation. Sometimes the issues are difficult to resolve, requiring more investigation and research. The issue process allows us to have a conversation about the problem and develop a satisfactory resolution.
- [Submitting a pull request](#) to add or change content is a more involved process. The following information outlines the tools, processes, and standards for submitting content to the documentation.

Prepare to make a contribution

Contributing to the documentation requires a GitHub account. Use the following checklist to install and configure the tools you need to make contributions.

1. [Sign up for GitHub](#)
2. [Install Git and Markdown tools](#)
3. [Install the Docs Authoring Pack](#)
4. [Install Posh-Git](#) ↗ - not required but recommended
5. [Set up a local Git repository](#)
6. [Review Git and GitHub fundamentals](#)

Get started writing docs

There are two ways to contribute changes to the documentation:

1. [Quick edits to existing docs](#) - Minor corrections, fixing typos, or small additions
2. [Full GitHub workflow for docs](#) - large changes, multiple versions, adding or changing images, or contributing new articles

Also, read the [Writing essentials](#) section of the centralized Contributor's Guide. Another excellent resource is the [Microsoft Writing Style Guide](#).

Minor corrections or clarifications to documentation and code examples in public repositories are covered by the [learn.microsoft.com Terms of Use](#).

Use the full GitHub workflow when you're making significant changes. If you're not an employee of Microsoft, our PR validation system adds a comment to the pull request asking you to sign the online [Contribution Licensing Agreement \(CLA\)](#). You must complete this step before we can review or accept your pull request. Signing the CLA is only required the first time you submit a PR in the repository. You might be asked to sign the CLA for each time you contribute to a new repository.

Code of conduct

All repositories that publish to Microsoft Learn adhere to the [Microsoft Open Source Code of Conduct](#) or the [.NET Foundation Code of Conduct](#). For more information, see the [Code of Conduct FAQ](#).

Next steps

The following articles cover information specific to PowerShell documentation. Where there's overlap with the guidance in the centralized Contributor's Guide, we call out how those rules differ for the PowerShell content.

Review the following documents:

- [Get started writing docs](#)
- [Markdown best practices](#)
- [PowerShell-Docs style guide](#)
- [How to file an issue](#)
- [Submitting a pull request](#)

Additional resources

- Editorial checklist
- How we manage issues
- How we manage pull requests

Get started contributing to PowerShell documentation

Article • 03/30/2025

This article is an overview of how to get started as a contributor to the PowerShell documentation.

PowerShell-Docs structure

There are three categories of content in the [PowerShell-Docs](#) repository:

- reference content
- conceptual content
- metadata and configuration files

Reference content

The reference content is the PowerShell cmdlet reference for the cmdlets that ship in PowerShell. The cmdlet [reference](#) is collected in versioned folders (like 5.1, 7.4, 7.5, and 7.6), which contain the reference for the modules that ship with PowerShell. This content is also used to create the help information displayed by the `Get-Help` cmdlet.

Conceptual content

The [conceptual documentation](#) isn't organized by version. All articles are displayed for every version of PowerShell.

 **Note**

Anytime a conceptual article is added, removed, or renamed, the TOC must be updated and deleted or renamed files must be redirected.

Metadata files

This project contains several types of metadata files. The metadata files control the behavior of our build tools and the publishing system. Only PowerShell-Docs maintainers and approved contributors are allowed to change these files. If you think that a meta file should be changed, open an issue to discuss the needed changes.

Meta files in the root of the repository

- `.*` - configuration files in the root of the repository
- `*.md` - Project documentation in the root of the repository
- `*.yml` - Project documentation in the root of the repository
- `.devcontainer/*` - devcontainer configuration files
- `.github/**/*` - GitHub templates, actions, and other meta files
- `.vscode/**/*` - VS Code extension configurations
- `assets/*` - contains downloadable files linked in the documentation
- `redir/*` - contain redirection mapping files
- `tests/*` - test tools used by the build system
- `tools/*` - other tools used by the build system

Meta files in the documentation set

- `reference/**/*.json` - docset configuration files
- `reference/**/*.yml` - TOC and other structured content files
- `reference/bread/*` - breadcrumb navigation configuration
- `reference/includes/*` - markdown include files
- `reference/mapping/*` - version mapping configuration
- `reference/**/media/**` - image files used in documentation
- `reference/module/*` - Module Browser page configuration

Creating new articles

A GitHub issue must be created for any new document you want to contribute. Check for existing issues to make sure you're not duplicating efforts. Assigned issues are considered to be `in progress`. If you wish to collaborate on an issue, contact the person assigned to the issue.

Similar to the PowerShell [RFC process](#), create an issue before you write the content. The issue ensures you don't waste time and effort on work that gets rejected by the PowerShell-Docs team. The issue allows us to consult with you on the scope of the content and where it fits in the PowerShell documentation. All articles must be included in the Table of Contents (TOC). The proposed TOC location should be included in the issue discussion.

Note

The publishing system autogenerates the TOC for reference content. You don't have to update the TOC.

Updating existing articles

Where applicable, cmdlet reference articles are duplicated across all versions of PowerShell maintained in this repository. When reporting an issue about a cmdlet reference or an `About_` article, list the versions of the article that have the problem.

Apply the appropriate change to each version of the file.

Localized content

The PowerShell documentation is written in English and translated into 17 other languages. The English content is stored in the GitHub repository named [MicrosoftDocs/PowerShell-Docs](#). Issues found in the translated content should be submitted to this repository.

All translations start from the English content first. We use both human and machine translation.

[+] Expand table

Translation method	Languages
Human translation	de-DE, es-ES, fr-FR, it-IT, ja-JP, ko-KR, pt-BR, ru-RU, zh-CN, zh-TW
Machine translation	cs-CZ, hu-HU, nl-NL, pl-PL, pt-PT, sv-SE, tr-TR

The content translated by machine translation might not always result in correct word choices and grammar. If you find an error in translation for any language, rather than in the technical details of the article, open an issue explaining why you think the translation is wrong.

Some translation issues can be fixed by changing the English source files. However, some issues can require updates to our internal translation system. For those cases, we must submit the issue to our internal localization team for review and response.

Next steps

There are two common ways of submitting changes in GitHub. Both methods are described in the central Contributor's Guide:

1. You can make [quick edits to existing documents](#) in the GitHub web interface.
2. Use the [full GitHub workflow](#) for adding new articles, updating multiple files, or other large changes.

Before starting any changes, you should create a fork of the PowerShell-Docs repository. The changes should be made in a working branch in your copy of the PowerShell-Docs. If you're using the [quick edit](#) method in GitHub, these steps are handled for you. If you're using the [full GitHub workflow](#), you must be set up to [work locally](#).

Both methods end with the creation of a Pull Request (PR). For more information and best practices, see [Submitting a pull request](#).

Contribute using GitHub Codespaces

Article • 03/30/2025

GitHub has a feature called [Codespaces](#) that you can use to contribute to the PowerShell documentation without having to install or configure any software locally. When you use a codespace, you get the same authoring tools the team uses for writing and editing.

You can use a codespace in your browser, making your contributions in VS Code hosted over the internet. If you have VS Code installed locally, you can connect to the codespace there too.

Available tools

When you use a codespace to contribute to the PowerShell documentation, your editor has these tools already available for you:

- [Markdownlint](#) for checking your Markdown syntax.
- [cSpell](#) for checking your spelling.
- [Vale](#) for checking your prose.
- The [Learn Authoring Pack](#) for inserting platform-specific syntax, previewing your contribution, and more.
- The [Reflow Markdown](#) extension for wrapping your Markdown as needed, making reading and editing easier.
- The [Table Formatter](#) extension for making your tables more readable without having to manually align columns.
- The [change-case](#) extension for converting the casing of your headings and prose.
- The [GitLens](#) extension for reviewing historical file changes.
- The [PowerShell](#) extension for interacting with PowerShell examples.
- The [Gremlins tracker for Visual Studio Code](#) for finding problematic characters in your Markdown.

Cost

GitHub Codespaces can be used for free up to 120 core-hours per month. The monthly usage is calculated across all your repositories, not just documentation.

For more information about pricing, see [About billing for GitHub Codespaces](#).

💡 Tip

If you're comfortable using containers and Docker, you can get the same experience by using the devcontainer defined for the PowerShell documentation repositories. There's no cost associated with using devcontainers. For more information, see the [Dev Containers tutorial](#).

Creating your GitHub Codespace

To create your GitHub Codespace for contributing to PowerShell documentation, follow these steps:

1. Open <https://github.com/codespaces> in your browser.
2. Select the "New codespace" button in the top right of the page.
3. In the "Create a new codespace" page, select the "Select a repository" button and type the name of the repository you want to contribute to, like `MicrosoftDocs/PowerShell-Docs`.
4. Leave all other settings as their default.
5. Select the "Create codespace" button.

After following these steps, GitHub creates a new codespace for that repository and sets it up for you. When the codespace is ready, the page refreshes and shows the web editor UI for the codespace. The UI is based on VS Code and works the same way.

Opening your GitHub Codespace

To open your GitHub Codespace in the browser, follow these steps:

1. Open <https://github.com/codespaces> in your browser.
2. The page lists your Codespaces. Find the created codespace for the repository you want to contribute to and select it.

After you select your codespace, GitHub opens it in the same window. From here, you're ready to contribute.

To open your GitHub Codespace in VS Code, follow the steps in [Using GitHub Codespaces in Visual Studio Code](#).

Authoring in your GitHub Codespace

Once you have your GitHub Codespace open in your browser or VS Code, contributing to the documentation follows the same process.

The rest of this article describes a selection of tasks you can do in your GitHub Codespace while writing or editing your contribution.

Extract a reference link

When you want to turn an inline link, like `[some text](destination.md)`, into a reference link like `[some text][01]`, select the link destination in the editor. Then you can either:

1. Right-click on the link destination and select "Refactor..." in the context menu.
2. Press `Ctrl + Shift + R`.

Either action raises the refactoring context menu. To replace the `(destination.md)` in the link with `[def]`, select **Extract to link definition** in the context menu. You can rename the definition by typing a name in.

For the PowerShell documentation, we use two-digit numerical reference link definitions, like `[01]` or `[31]`. Only use reference link definitions in about articles and conceptual documentation. Don't use reference link definitions in cmdlet reference documentation.

Fix prose style violations

When you review an article in your codespace, Vale automatically checks the article when you first open it and every time you save it. If Vale finds any style violations, it highlights them in the document with colored squiggles.

Hover over a violation to see more information about it.

To open a web page that explains the rule, select the rule's name in the hover information. To open the rule and review its implementation, select the rule's filename (ending in `.yml`).

If the rule supports a quick fix, you can select "Quick Fix..." in the hover information for the violation and apply one of the suggested fixes by selecting it from the context menu. You can also press `ctrl + .` when your cursor is on a highlighted problem to apply a quick fix if the rule supports it.

If the rule doesn't support quick fixes, read the rule's message and fix it if you can. If you're not sure how to fix it, the editors can make a suggestion when reviewing your PR.

Fix syntax problems

When you review an article in your codespace, Markdownlint automatically checks the article when you open it and as you type. If Markdownlint finds any syntax problems, it highlights them in the document with colored squiggles.

Hover over a violation to see more information about it. To open a web page that explains the rule, select the rule's ID in the hover information.

If the rule supports a quick fix, you can select "Quick Fix..." in the hover information for the violation and apply one of the suggested fixes by selecting it from the context menu. You can also press `ctrl + .` when your cursor is on a highlighted problem to apply a quick fix if the rule supports it.

If the rule doesn't support quick fixes, read the rule's message and fix it if you can. If you're not sure how to fix it, the editors can make a suggestion when reviewing your PR.

You can also apply fixes to all syntax violations in an article. To do so, open the command palette and type `Fix all supported markdownlint violations in the document`. As you type, the command palette filters the available commands. Select the "Fix all supported markdownlint violations in the document" command. When you do, Markdownlint updates the document to resolve any violations it has a quick fix for.

Format a table

To format a Markdown table, place your cursor in or on the table in your Markdown. Open the Command Palette and search for the `Table: Format Current` command. When you select that command, it updates the Markdown for your table to align and pad the table for improved readability.

It converts a table defined like this:

```
markdown
| foo | bar | baz |
|---|---|-|
| a | b | c |
```

Into this:

```
markdown
|   foo   |   bar   |   baz   |
| :---: | :---: | ---: |
|     a   |     b   |       c |
```

Insert an alert

The documentation uses [alerts](#) to make information more notable to a reader.

To insert an alert, you can, open the Command Palette and search for the [Learn: Alert](#) command. When you select that command, it opens a context menu. Select the alert type you want to add. When you do, the command inserts the alert's Markdown at your cursor in the document.

Make a heading use sentence casing

To convert a heading's casing, highlight the heading's text except for the leading `#` symbols, which set the heading level. When you have the text highlighted, open the Command Palette and search for the [Change case sentence](#) command. When you select that command, it converts the casing of the highlighted text.

You can also use the casing commands for any text in the document.

Open the Command Palette

You can use VS Code's [Command Palette](#) to run many helpful commands.

To open the Command Palette in the UI, select "View" in the top menu bar. Then select "Command Palette..." in the context menu.

To open the Command Palette with your keyboard, press the key combination for your operating system:

- Windows and Linux: `Ctrl + Shift + P`
- macOS: `Cmd + Shift + P`

Preview your contribution

To preview your contribution, open the Command Palette and search for the [Markdown: Open Preview](#) command. When you select that command, VS Code opens a preview of the active document. The preview's style matches the Learn platform.

Note

Site-relative and cross-reference links don't work in the preview.

Reflow your content

To limit the line lengths for a paragraph in a document, place your cursor on the paragraph. Then open the Command Palette and search for the `Reflow Markdown` command. When you select the command, it updates the current paragraph's line lengths to the configured length. For our repositories, that length is 99 characters.

When using this command for block quotes, make sure the paragraph in the block quote you're reflowing is surrounded by blank lines. Otherwise, the command reflows every paragraph together.

✖ Caution

Don't use this command when editing about articles. The lines in those documents must not be longer than 80 characters. There's currently no way for a repository to configure different line lengths by folder or filename, so reflow doesn't work for about article documents.

Review all problems in a document

To review all syntax and style rule violations in a document, open the Problems View.

To open the Problems View in the UI, select "View" in the top menu bar. Then select "Problems" in the context menu.

To open the Problems View with your keyboard, press the key combination for your operating system:

- Windows and Linux: `Ctrl + Shift + M`
- macOS: `Cmd + Shift + M`

The Problems View displays all errors, warnings, and suggestions for the open document. Select a problem to scroll to it in the document.

You can filter the problems by type or text matching.

Updating the `ms.date` metadata

To update the `ms.date` metadata for an article, open the Command Palette and search for the `Learn: Update "ms.date" Metadata Value` command. When you select the command, it updates the metadata to the current date.

Additional resources

The tasks and commands described in this article don't cover everything you can do with VS Code or the installed extensions.

For more information on using VS Code, see these articles:

- [Visual Studio Code Tips and Tricks ↗](#)
- [Basic Editing ↗](#)
- [Using Git source control in VS Code ↗](#)
- [Markdown and Visual Studio Code ↗](#)

For more information about the installed extensions, see their documentation:

- [change-case ↗](#)
- [GitLens ↗](#)
- [Gremlins tracker for Visual Studio Code ↗](#)
- [Learn Authoring Pack ↗](#)
- [markdownlint ↗](#)
- [Reflow Markdown ↗](#)
- [Table Formatter ↗](#)

Markdown best practices

Article • 04/29/2025

This article provides specific guidance for using Markdown in our documentation. It isn't a tutorial for Markdown. If you need a tutorial for Markdown, see this [Markdown cheatsheet](#).

The build pipeline that builds our documentation uses [markdig](#) to process the Markdown documents. Markdig parses the documents based on the rules of the latest [CommonMark](#) specification. OPS follows the CommonMark specification and adds some extensions for platform-specific features, such as tables and alerts.

The CommonMark specification is stricter about the construction of some Markdown elements. Pay close attention to the details provided in this document.

We use the [markdownlint](#) extension in VS Code to enforce our style and formatting rules. This extension is installed as part of the [Learn Authoring Pack](#).

Blank lines, spaces, and tabs

Blank lines also signal the end of a block in Markdown.

- Put a single blank between Markdown blocks of different types; for example, between a paragraph and a list or header.
- Don't use more than one blank line. Multiple blank lines render as a single blank line in HTML, therefore the extra blank lines are unnecessary.
- Don't use put multiple consecutive blank lines in a code block, consecutive blank lines break the code block.

Spacing is significant in Markdown.

- Remove extra spaces at the end of lines. Trailing spaces can change how Markdown renders.
- Always use spaces instead of tabs (hard tabs).

Titles and headings

Use [ATX headings](#) only (# style, as opposed to = or - style headers).

- Use sentence case - only proper nouns and the first letter of a title or header should be capitalized
- Include a single space between the # and the first letter of the heading
- Surround headings with single blank line

- Don't use more than one H1 per document
 - It should be the first header
 - It should match the title of the article
- Increment header levels by one - don't skip levels
- Limit depth to H3 or H4
- Avoid using bold or other markup in headers

Limit line length to 100 characters

For conceptual articles and cmdlet reference, limit lines to 100 characters. For `about_` articles, limit the line length to 79 characters. Limiting the line length improves the readability of `git` diffs and history. It also makes it easier for other writers to make contributions.

Use the [Reflow Markdown](#) extension in VS Code to reflow paragraphs to fit the prescribed line length.

Some content types can't be easily reflowed. For example, the code inside of code blocks can also be difficult to reflow, depending on the content and the code language. You can't reflow a table. In these cases, use your best judgment to keep the content as close to the 100-character limit as possible.

Emphasis

For emphasis, Markdown supports bold and italics. Markdown allows you to use either `*` or `_` to mark the emphasis. However, to be consistent and show intent:

- Use `**` for bold
- Use `_` for italics

Following this pattern makes it easier for others to understand the intent of the markup when there's a mix of bold and italics in a document.

Lists

If your list has multiple sentences or paragraphs, consider using a sublevel header rather than a list.

Surround lists with a single blank line.

Unordered lists

- Don't end list items with a period unless they contain multiple sentences.
- Use the hyphen character (-) for list item bullets to avoid confusion with emphasis markup that uses the asterisk (*).
- To include a paragraph or other elements under a bullet item, insert a line break and align indentation with the first character after the bullet.

For example:

Markdown

This is a list that contains child elements under a bullet item.

- First bullet item
 - Sentence explaining the first bullet.
 - Child bullet item
 - Sentence explaining the child bullet.
 - Second bullet item
 - Third bullet item

This is a list that contains child elements under a bullet item.

- First bullet item
 - Sentence explaining the first bullet.
 - Child bullet item
 - Sentence explaining the child bullet.
 - Second bullet item
 - Third bullet item

Ordered lists

- All items in a numbered list should use the number 1 rather than incrementing each item.
 - Markdown rendering increments the value automatically.
 - This makes reordering items easier and standardizes the indentation of subordinate content.
- To include a paragraph or other elements under a numbered item, align indentation with the first character after the item number.

For example:

Markdown

1. For the first element, insert a single space after the `1`. Long sentences should be wrapped to the next line and must line up with the first character after the numbered list marker.

To include a second element, insert a line break after the first and align indentations. The indentation of the second element must line up with the first character after the numbered list marker.

1. The next numbered item starts here.

The resulting Markdown is rendered as follows:

1. For the first element, insert a single space after the `1`. Long sentences should be wrapped to the next line and must line up with the first character after the numbered list marker.

To include a second element, insert a line break after the first and align indentations. The indentation of the second element must line up with the first character after the numbered list marker.
2. The next numbered item starts here.

Images

The syntax to include an image is:

Markdown

```
![alt text](<code>folderPath</code>)
```

Example:

```
![Introduction](<code>./media/overview/Introduction.png</code>)
```

Where `alt text` is a brief description of the image and `<FolderPath>` is a relative path to the image.

- Alternate text is required to support screen readers for the visually impaired.
- Images should be stored in a `media/<article-name>` folder within the folder containing the article.

- Create a folder that matches the filename of your article under the `media` folder. Copy the images for that article to that new folder.
- Don't share images between articles.
 - If an image is used by multiple articles, each folder must have a copy of that image.
 - This prevents a change to an image in one article from affecting another article.

The following image file types are supported: `*.png`, `*.gif`, `*.jpeg`, `*.jpg`, `*.svg`

Markdown extension - Alert boxes

The [Learn Authoring Pack](#) contains tools that support features unique to our publishing system. Alerts are a Markdown extension to create blockquotes that render with colors and icons highlighting the significance of the content. The following alert types are supported:

Markdown

```
> [!NOTE]
> Information the user should notice even if skimming.

> [!TIP]
> Optional information to help a user be more successful.

> [!IMPORTANT]
> Essential information required for user success.

> [!CAUTION]
> Negative potential consequences of an action.

> [!WARNING]
> Dangerous certain consequences of an action.
```

These alerts look like this on Microsoft Learn:

Note block

!Note

Information the user should notice even if skimming.

Tip block

💡Tip

Optional information to help a user be more successful.

Important block

Important

Essential information required for user success.

Caution block

Caution

Negative potential consequences of an action.

Warning block

Warning

Dangerous certain consequences of an action.

Markdown extension - Tables

A table is an arrangement of data with rows and columns consisting of:

- A single header row
- A delimiter row separating the header from the data
- Zero or more data rows

Each row consists of cells containing arbitrary text separated by pipes (`|`). A leading and trailing pipe is also recommended for clarity. Spaces between pipes and cell content are trimmed. Block-level elements can't be inserted in a table. All content of a row must be on a single line.

The delimiter row consists of cells whose only content are hyphens (`-`), and optionally, a leading or trailing colon (`:`), or both, to indicate left, right, or center alignment respectively.

For small tables, consider using a list instead. Lists are:

- Easier to maintain and read
- Can be reflowed to fit within the 100-character line limit
- More accessible for users that use screen readers for visual assistance

For more information, see *Tables* section of [Markdown reference for Microsoft Learn](#).

Hyperlinks

- Hyperlinks must use Markdown syntax `[friendlyname](url-or-path)`.
- The publishing system supports three types of links:
 - URL links
 - File links
 - Cross-reference links
- All URLs to external websites should use HTTPS unless that isn't valid for the target site.
- Links must have a friendly name, usually the title of the linked article
- Avoid using backticks, bold, or other markup inside the brackets of a hyperlink.
- Bare URLs can be used when you're documenting a specific URI but must be enclosed in backticks. For example:

Markdown

```
By default, if you don't specify this parameter, the DMTF standard resource
URI
`http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/` is used and the class
name is appended to it.
```

- Use [link references](#) where allowed. Link references within paragraphs make the paragraphs more readable.

Link references divide a Markdown link into two parts:

- the link reference - `[friendlyname][id]`
- the link definition - `[id]: url-or-path`

URL-type Links

- URL links to other articles on `learn.microsoft.com` must use site-relative paths. The simplest way to create a site-relative link is to copy the URL from your browser then remove `https://learn.microsoft.com/en-us` from the value you paste into markdown.
- Don't include locales in URLs on Microsoft properties (remove `/en-us` from URL) or Wikipedia links.
- Remove any unnecessary query parameters from the URL. Examples that should be removed:
 - `?view=powershell-5.1` - used to link to a specific version of PowerShell

- `?redirectedfrom=MSDN` - added to the URL when you get redirected from an old article to its new location
- If you need to link to a specific version of a document, you must add the `&preserve-view=true` parameter to the query string. For example: `?view=powershell-5.1&preserve-view=true`
- On Microsoft sites, URL links don't contain file extensions (for example, no `.md`)

File-type links

- A file link is used to link from one reference article to another, or from one conceptual article to another in the same docset.
 - If you need to link from a conceptual article to a reference article you must use a URL link.
 - If you need to link to an article in another docset or across repositories you must use a URL link.
- Use relative filepaths (for example: `../folder/file.md`)
- All file paths use forward-slash (`/`) characters
- Include the file extension (for example, `.md`)

Cross-reference links

Cross-reference links are a special feature supported by the publishing system. You can use cross-reference links in conceptual articles to link to .NET API or cmdlet reference.

- For links to .NET API reference, see [Use links in documentation](#) in the central Contributor Guide.
- Links to cmdlet reference have the following format: `xref:<module-name>.<cmdlet-name>`. For example, to link to the `Get-Content` cmdlet in the `Microsoft.PowerShell.Management` module.

`[Get-Content](xref:Microsoft.PowerShell.Management.Get-Content)`

Deep linking

Deep linking is allowed on both URL and file links.

- The anchor text must be lowercase
- Add the anchor to the end of the target path. For example:
 - `[about_Splatting](about_Splatting.md#splatting-with-arrays)`

- [custom key bindings]
(https://code.visualstudio.com/docs/getstarted/keybindings#_custom-keybindings-for-refactorings)

For more information, see [Use links in documentation](#).

Code spans

Code spans are used for inline code snippets within a paragraph. Use single backticks to indicate a code span. For example:

Markdown

```
PowerShell cmdlet names use the `Verb-Noun` naming convention.
```

This example renders as:

PowerShell cmdlet names use the `Verb-Noun` naming convention.

Code blocks

Code blocks are used for command examples, multi-line code samples, query languages, and outputs. There are two ways to indicate a section of text in an article file is a code block: by fencing it in triple-backticks (```) or by indenting it.

Never use indentation because it's too easy to get wrong and it may be difficult for another writer to understand your intent when they need to edit your article.

Fenced code blocks can include an optional tag that indicates the language syntax contained in the block. The publishing platform supports a list of [language tags](#). The language tag is used to provide syntax highlighting when the article is rendered on the webpage. The language tag is not case-sensitive, but you should use lowercase except for a few special cases.

- Code fences without tags can be used for syntax blocks or other types of content where syntax highlighting is not required.
- When showing output from a command, use a tagged code fence with the language tag `Output`. The rendered box is labeled as **Output** and doesn't have syntax highlighting.
- If the output is in a specific supported language, use the appropriate language tag. For example, many commands output JSON, so use the `json` tag.
- If you use an unsupported language tag, the code block will render without syntax highlighting. The language tag becomes the label for the rendered code box on the webpage. Capitalize the tag so that the label is capitalized on the webpage.

Next steps

[PowerShell style guide](#)

PowerShell-Docs style guide

Article • 03/30/2025

This article provides style guidance specific to the PowerShell-Docs content. It builds on the information outlined in the [Overview](#).

Formatting command syntax elements

Use the following rules to format elements of the PowerShell language when the elements are used in a sentence.

- Always use the full name for cmdlets and parameters in the proper Pascal case
- Only use an alias when you're specifically demonstrating the alias
- PowerShell keywords and operators should be all lowercase
- The following items should be formatted using **bold** text:
 - Type names
 - Class names
 - Property names
 - Parameter names
 - By default, use the parameter without the hyphen prefix.
 - Use parameter name with the hyphen if you're illustrating syntax. Wrap the parameter in backticks.

For example:

markdown

```
The parameter's name is **Name**, but it's typed as ` -Name` when used on  
the command  
line as a parameter.
```

- The following items should be formatted using backticks (`):
 - Property and parameter values
 - Type names that use the bracketed style - For example: [System.Io.FileInfo]

- o Referring to characters by name. For example: Use the asterisk character (*) to as a wildcard.
- o Language keywords and operators
- o Cmdlet, function, and script names
- o Command and parameter aliases
- o Method names - For example: The `Tostring()` method returns a string representation of the object
- o Variables
- o Native commands
- o File and directory paths
- o Inline command syntax examples - See [Markdown for code samples](#)

This example shows some backtick examples:

markdown

The following code uses `Get-ChildItem` to list the contents of `C:\Windows` and assigns the output to the `\$files` variable.

```
```powershell
$files = Get-ChildItem C:\Windows
```
```

This example shows command syntax inline:

markdown

To start the spooler service on a remote computer named DC01, you type:
`sc.exe \\DC01 start spooler`.

Including the file extension ensures that the correct command is executed according to PowerShell's command precedence.

Markdown for code samples

Markdown supports two different code styles:

- **Code spans (inline)** - marked by a single backtick (`) character. Used within a paragraph rather than as a standalone block.
- **Code blocks** - a multi-line block surrounded by triple-backtick (``` strings. Code blocks can also have a language label following the backticks. The language label enables syntax highlighting for the contents of the code block.

All code blocks should be contained in a code fence. Never use indentation for code blocks. Markdown allows this pattern but it can be problematic and should be avoided.

A code block is one or more lines of code surrounded by a triple-backtick (``` code fence. The code fence markers must be on their own line before and after the code sample. The opening marker can have an optional language label. The language label enables syntax highlighting on the rendered webpage.

For a full list of supported language tags, see [Fenced code blocks](#) in the centralized contributor guide.

Publishing also adds a **Copy** button that can copy the contents of the code block to the clipboard. This allows you to paste the code into a script to test the code sample. However, not all examples are intended to be run as written. Some code blocks are basic illustrations of PowerShell concepts.

There are three types code blocks used in our documentation:

1. Syntax blocks
2. Illustrative examples
3. Executable examples

Syntax code blocks

Syntax code blocks are used to describe the syntactic structure of a command. Don't use a language tag on the code fence. This example illustrates all the possible parameters of the `Get-Command` cmdlet.

markdown

```
```
Get-Command [-Verb <String[]>] [-Noun <String[]>] [-Module <String[]>]
 [-FullyQualifiedModule <ModuleSpecification[]>] [-TotalCount <Int32>] [-Syntax]
 [-ShowCommandInfo] [[-ArgumentList] <Object[]>] [-All] [-ListImported]
 [-ParameterName <String[]>] [-ParameterType <PSTypeName[]>] [<CommonParameters>]
````
```

This example describes the `for` statement in generalized terms:

```
markdown
```

```
```  
for (<init>; <condition>; <repeat>)
{<statement list>}
```
```

Illustrative examples

Illustrative examples are used to explain a PowerShell concept. You should [Avoid using PowerShell prompts in examples](#) whenever possible. However, illustrative examples aren't meant to be copied and pasted for execution. They're most commonly used for simple examples that are easy to understand. You may include the PowerShell prompt and example output.

Here's a simple example illustrating the PowerShell comparison operators. In this case, we don't intend the reader to copy and run this example. Notice that this example uses `PS>` as a simplified prompt string.

```
markdown
```

```
```powershell  
PS> 2 -eq 2
True

PS> 2 -eq 3
False

PS> 1,2,3 -eq 2
2

PS> "abc" -eq "abc"
True

PS> "abc" -eq "abc", "def"
False

PS> "abc", "def" -eq "abc"
abc
```
```

Executable examples

Complex examples, or examples that are intended to be copied and executed, should use the following block-style markup:

markdown

```
```powershell
<Your PowerShell code goes here>
```
```

The output displayed by PowerShell commands should be enclosed in an **Output** code block to prevent syntax highlighting. For example:

markdown

```
```powershell
Get-Command -Module Microsoft.PowerShell.Security
```

```Output
CommandType Name Version Source
----- ----
Cmdlet ConvertFrom-SecureString 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet ConvertTo-SecureString 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Get-Acl 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Get-AuthenticodeSignature 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Get-CmsMessage 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Get-Credential 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Get-ExecutionPolicy 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Get-PfxCertificate 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet New-FileCatalog 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Protect-CmsMessage 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Set-Acl 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Set-AuthenticodeSignature 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Set-ExecutionPolicy 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Test-FileCatalog 3.0.0.0 Microsoft.PowerShell.Security
Cmdlet Unprotect-CmsMessage 3.0.0.0 Microsoft.PowerShell.Security
```
```

```

The **Output** code label isn't an official *language* supported by the syntax highlighting system. However, this label is useful because our publishing system adds the **Output** label to the code box frame on the web page. The **Output** code box has no syntax highlighting.

## Coding style rules

### Avoid line continuation in code samples

Avoid using line continuation characters (`) in PowerShell code examples. Backtick characters are difficult to see and can cause problems if there are extra spaces at the end of the line.

- Use PowerShell [splatting](#) to reduce line length for cmdlets that have several parameters.

- Take advantage of PowerShell's natural line break opportunities, like after pipe ( | ) characters, opening braces ( { ), parentheses ( ( ), and brackets ( [ ).

## Avoid using PowerShell prompts in examples

Use of the prompt string is discouraged and should be limited to scenarios that are meant to illustrate command-line usage. For most of these examples, the prompt string should be PS>. This prompt is independent of OS-specific indicators.

Prompts are required in examples to illustrate commands that alter the prompt or when the path displayed is significant to the scenario. The following example illustrates how the prompt changes when using the Registry provider.

```
PowerShell

PS C:\> cd HKCU:\System\
PS HKCU:\System\> dir

 Hive: HKEY_CURRENT_USER\System

Name Property
---- -----
CurrentControlSet
GameConfigStore GameDVR_Enabled : 1
 GameDVR_FSEBehaviorMode : 2
 Win32_AutoGameModeDefaultProfile : {2, 0, 1, 0...}
 Win32_GameModeRelatedProcesses : {1, 0, 1, 0...}
 GameDVR_HonorUserFSEBehaviorMode : 0
 GameDVR_DXGIIHonorFSEWindowsCompatible : 0
```

## Don't use aliases in examples

Use the full name of all cmdlets and parameters unless you're specifically documenting the alias. Cmdlet and parameter names must use the proper [Pascal-cased](#) names.

## Using parameters in examples

Avoid using positional parameters. To reduce the chance of confusion, you should include the parameter name in an example, even if the parameter is positional.

## Formatting cmdlet reference articles

Cmdlet reference articles have a specific structure. [PlatyPS](#) defines this structure. PlatyPS generates the cmdlet help for PowerShell modules in Markdown. After you edit the Markdown files, PlatyPS can create the MAML help files used by the `Get-Help` cmdlet.

PlatyPS has a schema that expects a specific structure for cmdlet reference. The PlatyPS [schema document](#) describes this structure. Schema violations cause build errors that must be fixed before we can accept your contribution.

- Don't remove any of the ATX header structures. PlatyPS expects a specific set of headers in a specific order.
- The H2 **INPUTS** and **OUTPUTS** headers must have an H3 type. If the cmdlet doesn't take input or return a value, then use the value `None` for the H3.
- Inline code spans can be used in any paragraph.
- Fenced code blocks are only allowed in the **EXAMPLES** section.

In the PlatyPS schema, **EXAMPLES** is an H2 header. Each example is an H3 header. Within an example, the schema doesn't allow code blocks to be separated by paragraphs. The schema only allows the following structure:

```
Example X - Title sentence

0 or more paragraphs
1 or more code blocks
0 or more paragraphs.
```

Number each example and add a brief title.

For example:

```
markdown

Example 1: Get cmdlets, functions, and aliases

This command gets the PowerShell cmdlets, functions, and aliases that are
installed on the
computer.

```powershell
Get-Command
```

Example 2: Get commands in the current session

```powershell
```

```
Get-Command -ListImported  
...
```

Formatting About_ files

About_* files are written in Markdown but are shipped as plain text files. We use [Pandoc](#) to convert the Markdown to plain text. About_* files are formatted for the best compatibility across all versions of PowerShell and with the publishing tools.

Basic formatting guidelines:

- Limit paragraph lines to 80 characters
- Limit code blocks to 76 characters
- Limit blockquotes and alerts to 78 characters
- When using these special meta-characters \, \$, and <:
 - Within a header, these characters must be escaped using a leading \ character or enclosed in code spans using backticks (`)
 - Within a paragraph, these characters should be put into code spans. For example:

```
markdown  
  
### The purpose of the \$foo variable  
  
The `\$foo` variable is used to store ...
```

- Markdown tables
 - For About_* articles, tables must fit within 76 characters
 - If the content doesn't fit within 76 character limit, use bullet lists instead
 - Use opening and closing | characters on each line

Next steps

[Editorial checklist](#)

Editor's checklist

Article • 03/30/2025

This article contains a summarized list of rules for writing or editing PowerShell documentation. See other articles in the Contributor's Guide for detailed explanations and examples of these rules.

Metadata

- `ms.date`: must be in the format MM/DD/YYYY
 - Change the date when there's a significant or factual update
 - Reorganizing the article
 - Fixing factual errors
 - Adding new information
 - Don't change the date if the update is insignificant
 - Fixing typos and formatting
- `title`: unique string of 43-59 characters long (including spaces)
 - Don't include site identifier (it's autogenerated)
 - Use sentence case - capitalize only the first word and any proper nouns
- `description`: 115-145 characters including spaces - this abstract displays in the search result

Formatting

- Backtick syntax elements that appear, inline, within a paragraph
 - Cmdlet names `Verb-Noun`
 - Variable `$counter`
 - Syntactic examples `Verb-Noun -Parameter`
 - File paths `C:\Program Files\PowerShell`, `/usr/bin/pwsh`
 - URLs that aren't meant to be clickable in the document
 - Property or parameter values
- Use bold for property names, parameter names, class names, module names, entity names, object, or type names
 - Bold is used for semantic markup, not emphasis
 - Bold - use asterisks `**`
- Italic - use underscore `_`
 - Only used for emphasis, not for semantic markup
- Line breaks at 100 columns (or at 80 for `about_Topic`)
- No hard tabs - use spaces only

- No trailing spaces on lines
- PowerShell keywords and operators should be all lowercase
- Use proper (Pascal) casing for cmdlet names and parameters

Headers

- Start with H1 first - only one H1 per article
- Use [ATX Headers](#) only
- Use sentence case for all headers
- Don't skip levels - no H3 without an H2
- Limit header depth to H3 or H4
- Add blank lines before and after
- Don't add or remove headers - PlatyPS enforces specific headers in its schema

Code blocks

- Add blank lines before and after
- Use tagged code fences - **powershell**, **Output**, or other appropriate language ID
- Use untagged code fence for syntax blocks
- Put output in separate code block except for basic examples where you don't intend for the reader to use the **Copy** button
- See list of [supported languages](#)

Lists

- Indent properly
- Add blank lines before first item and after last item
- Use hyphen (-) for bullets not asterisk (*) to reduce confusion with emphasis
- Use 1. for all items in a numbered list

Terminology

- Use *PowerShell* vs. *Windows PowerShell*
- See [Product Terminology](#)

Cmdlet reference examples

- Must have at least one example in cmdlet reference
- Examples should be only enough code to demonstrate the usage

- PowerShell syntax
 - Use full names of cmdlets and parameters - no aliases
 - Use splatting for parameters when the command line gets too long
 - Avoid using line continuation backticks - only use when necessary
- Remove or simplify the PowerShell prompt (`PS>`) except where required for the example
- Cmdlet reference example must follow the following PlatyPS schema

markdown

```
### Example 1 - Descriptive title

Zero or more short descriptive paragraphs explaining the context of the
example followed by one or
more code blocks. Recommend at least one and no more than two.

```powershell
... one or more PowerShell code statements ...
```

```Output
Example output of the code above.
```

Zero or more optional follow up paragraphs that explain the details of the
code and output.
```

- don't put paragraphs between the code blocks. All descriptive content must come before or after the code blocks.

Linking to other documents

- When linking outside the docset or between cmdlet reference and conceptual
 - Use site-relative URLs when linking to Microsoft Learn (remove `https://learn.microsoft.com/en-us`)
 - don't include locales in URLs on Microsoft properties (remove `/en-us` from URL)
 - All URLs to external websites should use HTTPS unless that's not valid for the target site
- When linking within the docset
 - Use the relative filepath (`../folder/file.md`)
- All paths use forward-slash (/) characters
- Image links should have unique alt text

Product terminology and branding guidelines

Article • 03/30/2025

When you write about any product, it's important to use the proper product names and terminology. This guide defines product names and terminology related to PowerShell. Note the capitalization of specific words or use cases.

PowerShell (collective name)

Use **PowerShell** to describe the scripting language and an interactive shell.

PowerShell (product name)

The cross-platform version of PowerShell built on .NET (core), rather than the .NET Framework. PowerShell can be installed on Windows, Linux, and macOS.

PowerShell Core (product deprecated)

The name used for PowerShell v6, built on .NET Core. This name shouldn't be used.

Windows PowerShell (product name)

The version of PowerShell that ships in Windows, which requires the full .NET Framework.

Guidelines

- First mention - use "Windows PowerShell"
- Subsequent mentions - Use "PowerShell" unless the use case requires "Windows PowerShell" to be more specific:

In PowerShell, the `Invoke-WebRequest` cmdlet returns
`BasicHtmlWebResponseObject`

In Windows PowerShell, the `Invoke-WebRequest` cmdlet returns
`HtmlWebResponseObject`

PowerShell modules

PowerShell modules are add-ons that contain PowerShell cmdlets to manage specific products or services.

For example:

- Azure PowerShell
- Az.Accounts module
- Windows management module
- Hyper-V module
- Microsoft Graph PowerShell SDK
- Exchange PowerShell

Guidelines

- Always use the collective name or the more specific module name when referring to a PowerShell module
- Never refer to a module as "PowerShell"

Azure PowerShell (collective name)

The branded group of products containing PowerShell modules used to manage Azure.

There are several versions of Azure PowerShell products available. Each product contains multiple named modules.

Guidelines

- Use "Azure PowerShell" as the collective name for the product
- Always use the collective name, never just "PowerShell"
- Use the more specific product name when referring to a specific version

Az PowerShell (product name)

The currently supported collection of modules for managing Azure resources with PowerShell.

AzureRM PowerShell (product name)

The previous generation of modules that use the Azure Resource Manager (ARM) model for managing Azure resources. This product is deprecated, no longer maintained or supported, and not recommended.

Azure Service Management PowerShell (product name)

The earliest collection of modules for managing legacy Azure resources that use Azure Service Manager (ASM) APIs. This legacy PowerShell module isn't recommended when creating new resources since ASM is scheduled for retirement.

Azure-related PowerShell modules

These products are used to manage Azure resources but aren't part of the Azure PowerShell collective product. They should never be described using the "Azure PowerShell" collective name.

- Azure Information Protection PowerShell
- Azure Deployment Manager PowerShell
- Azure Elastic Database Jobs PowerShell
- Azure Service Fabric PowerShell
- Azure Stack PowerShell
- Microsoft Graph PowerShell SDK
- Microsoft Entra PowerShell

Guidelines

- Always use the full proper name of the product or the specific PowerShell module name

Other PowerShell-related products

Visual Studio Code (VS Code)

This is Microsoft's free, open source editor.

Guidelines

- First mention - use the full name
- Subsequent mentions - you can use "VS Code"
- Never use "VSCode"

PowerShell Extension for Visual Studio Code

The extension turns VS Code into the preferred IDE for PowerShell.

Guidelines

- First mention - use the full name
- Subsequent mentions - you can use "PowerShell extension"

How to file a PowerShell-Docs issue

Article • 03/30/2025

There are two ways to create an issue:

1. Use the feedback controls at the bottom of the page.
2. File an issue in GitHub directly

Using the feedback controls

For a full description of the feedback controls, see the Docs Team blog announcing this [feature](#).

At the bottom of most pages on `learn.microsoft.com`, there are two feedback buttons. One is a link for product feedback and one is for documentation feedback. The documentation feedback requires a GitHub account. Clicking the button takes you in GitHub and presents an issue template. Enter your feedback and submit the form.

Note

The feedback tool not a support channel. It's a way to ask questions to clarify documentation or to report errors and omissions. If you need technical support, see [Community resources](#).

Filing issues on GitHub

To file a GitHub issue directly, you can select the [New issue ↗](#) button in the PowerShell-Docs repository. Select the **Get started** button for the issue you want to create. The GitHub issue template helps you provide the information needed to address the problem you're reporting.

To avoid duplication, search the existing issues to see if someone else has already reported it. If you find an existing issue, you can add your comments, related questions, or answers.

Next steps

See [Get started writing](#).

Additional resources

[How we manage issues](#)

How to submit pull requests

Article • 03/30/2025

To make changes to content, submit a pull request (PR) from your fork. A pull request must be reviewed before it can be merged. For best results, review the [editorial checklist](#) before submitting your pull request.

Using git branches

The default branch for PowerShell-Docs is the `main` branch. Changes made in working branches are merged into the `main` branch before then being published. The `main` branch is merged into the `live` branch every weekday at 3:00 PM (Pacific Time). The `live` branch contains the content that is published to learn.microsoft.com.

Before starting any changes, create a working branch in your local copy of the PowerShell-Docs repository. When working locally, be sure to synchronize your local repository before creating your working branch. The working branch should be created from an up-to-date copy of the `main` branch.

All pull requests should target the `main` branch. Don't submit changes to the `live` branch. Changes made in the `main` branch get merged into `live`, overwriting any changes made to `live`.

Make the pull request process work better for everyone

The simpler and more focused you can make your PR, the faster it can be reviewed and merged.

Avoid pull requests that update large numbers of files or contain unrelated changes

Avoid creating PRs that contain unrelated changes. Separate minor updates to existing articles from new articles or major rewrites. Work on these changes in separate working branches.

Bulk changes create PRs with large numbers of changed files. Limit your PRs to a maximum of 50 changed files. Large PRs are difficult to review and are more prone to contain errors.

Renaming or deleting files

There must be an issue associated with the PR when you rename or delete files. That issue must discuss the need to rename or delete the files.

Avoid mixing content additions or changes with file renames and deletes. Any file that you rename or delete must be added to the appropriate redirection file. When possible, update any files that link to the renamed or deleted content, including any TOC files.

Avoid editing repository configuration files

Avoid modifying repository configuration files. Limit your changes where possible to the Markdown content files and any supporting image files needed for the content.

Incorrect modifications to repository configuration files can break the build, introduce vulnerabilities or accessibility issues, or violate organizational standards. Repository configuration files are any files that match one or more of these patterns:

- `*.yml`
- `.github/**`
- `.localization-config`
- `.openpublishing*`
- `LICENSE*`
- `reference/docfx.json`
- `reference/mapping/**`
- `tests/**`
- `ThirdPartyNotices`
- `tools/**`

For safety and security, don't change these files. If you think one of these files should be changed, [file an issue ↗](#). After the maintainers triage the issue, they'll make the appropriate changes.

Use the PR template

When you create a PR, a template is automatically inserted into the PR body for you. It looks like this:

```
Markdown

# PR Summary

<!--
    Delete this comment block and summarize your changes and list
    related issues here. For example:
-->
```

```

This changes fixes problem X in the documentation for Y.

- Fixes #1234
- Resolves #1235
-->

## PR Checklist

<!--
These items are mandatory. For your PR to be reviewed and merged,
ensure you have followed these steps. As you complete the steps,
check each box by replacing the space between the brackets with an
x or by clicking on the box in the UI after your PR is submitted.
-->

- [ ] **Descriptive Title:** This PR's title is a synopsis of the changes it
proposes.
- [ ] **Summary:** This PR's summary describes the scope and intent of the change.
- [ ] **Contributor's Guide:** I have read the [contributors guide][contrib].
- [ ] **Style:** This PR adheres to the [style guide][style].
```

<!--

If your PR is a work in progress, please mark it as a draft or prefix it with "(WIP)" or "WIP:"

This helps us understand whether or not your PR is ready to review.

-->

[contrib]: /powershell/scripting/community/contributing/overview
[style]: /powershell/scripting/community/contributing/powershell-style-guide

In the "PR Summary" section, write a short summary of your changes and list any related issues by their issue number, like #1234. If your PR fixes or resolves the issue, use GitHub's [autoclose](#) feature so the issue is automatically closed when your PR is merged.

Review the items in the "PR Checklist" section and check them off as you complete each one. You must follow the directions and check each item for the team to approve your PR.

If your PR is a work-in-progress, set it to [draft mode](#) or prefix your PR title with [WIP](#).

Expectations Comment

After you submit your PR, a bot will comment on your PR. The comment provides resources and sets expectations for the rest of the process. We might update this comment periodically, so always review the comment, even if this isn't your first contribution.



github-actions bot commented 9 days ago

...

Expectations

Thanks for your submission! Here's a quick note to provide you with some context for what to expect from the docs team and the process now that you've submitted a PR. Even if you've contributed to this repo before, we strongly suggest reading this information; it might have changed since you last read it.

To see our process for reviewing PRs, please read our [editor's checklist](#) and process for [managing pull requests](#) in particular. Below is a brief, high-level summary of what to expect, but our contributor guide has expanded details.

The docs team begins to review your PR if you [request them to](#) or if your PR meets these conditions:

- It is not a [draft PR](#).
- It does not have a [WIP](#) prefix in the title.
- It passes validation and build steps.
- It does not have any merge conflicts.
- You have checked every box in the [PR Checklist](#), indicating you have completed all required steps and marked your PR as [Ready to Merge](#).

You can [always](#) request a review at any stage in your authoring process, the docs team is here to help! You do not need to submit a fully polished and finished draft; the docs team can help you get content ready for merge.

While reviewing your PR, the docs team may make suggestions, write comments, and ask questions. When all requirements are satisfied, the docs team marks your PR as [Approved](#) and merges it. Once your PR is merged, it is included the next time the documentation is published. For this project, the documentation is published daily at 3 p.m. Pacific Standard Time (PST).

Docs PR validation service

The Docs PR validation service is a GitHub app that runs validation rules on your changes. You must fix any errors or warnings reported by the validation service.

The following steps outline the validation behavior:

1. You submit a PR.
2. In the GitHub comment that indicates the status of the "checks" enabled on the repository. In this example, there are two checks enabled, "Commit Validation" and "OpenPublishing.Build":

Validation status: errors

| File | Status | Preview URL | Details |
|---|--|-------------|-------------------------|
| reference/powershell-6/Microsoft.PowerShell.Management/Set-Content.md |  Error | | Details |
| |  Error | | Details |

[reference/powershell-6/Microsoft.PowerShell.Management/Set-Content.md](#)

- [Error] Unable to load file: powershell-6/Microsoft.PowerShell.Management/Set-Content.md via processor: AzureCliDocumentProcessor.
- [Error] :70:(338) `powershell
Get-Content test.xml | Set-Content ...`
Expect Heading
- [Warning] 'W:\ewzo-s\gallery/index.md' would be overwritten by redirection rule configured in master redirection file .openpublishing.redirection.json. Please remove the original file to resolve this warning.
- [Warning] 'W:\ewzo-s\jea/index.md' would be overwritten by redirection rule configured in master redirection file .openpublishing.redirection.json. Please remove the original file to resolve this warning.

For more details, please refer to the [build report](#).

Note: If you changed an existing file name or deleted a file, broken links in other files to the deleted or renamed file are listed only in the full build report.

The build can pass even if commit validation fails.

3. Select **Details** for more information. The **Details** page shows all the validation checks that failed and includes information about how to fix the issues.
4. When validation succeeds, the following comment is added to the PR:

Validation status: passed

| File | Status | Preview URL | Details |
|---|---|---|---------|
| reference/docs-conceptual/PowerShell-Scripting.md |  Succeeded | View (powershell-3.0)
View (powershell-4.0)
View (powershell-5.0)
View (powershell-5.1)
View (powershell-6) | |

For more details, please refer to the [build report](#).

Note: If you changed an existing file name or deleted a file, broken links in other files to the deleted or renamed file are listed only in the full build report.

Note

If you're an external contributor (not a Microsoft employee), you don't have access to the detailed build reports or preview links.

When the PR is reviewed, you might be asked to make changes or fix validation warning messages. The PowerShell-Docs team can help you understand validation errors and editorial requirements.

GitHub Actions

Several different GitHub Actions run against your changes to validate and provide context for you and the reviewers.

Checklist verification

If your PR isn't in [draft mode](#) and isn't prefixed with `WIP`, a GitHub Action inspects your PR to verify that you selected every item in the PR template's checklist. The maintainers won't review or merge your PR until you complete the checklist. The checklist items are mandatory.

Authorization verification

If your PR targets the `live` branch or modifies any repository configuration files, a GitHub Action checks your permissions to verify that you're authorized to submit those changes.

Only repository administrators are authorized to target the `live` branch or modify repository configuration files.

Versioned content change reporting

If your PR adds, removes, or modifies any versioned content a GitHub Action analyzes your changes and writes a report summarizing the types of changes made to versioned content.

This report can show if there are other versions of the files that you need to update in this PR.

To find the versioned content report for your PR:

1. Selecting the "Checks" tab on your PR page.
2. Select the "Reporting" job from the list of jobs.
3. Select the "..." button in the top right.
4. Select "View job summary."

Versioned Content Change Report

Pull Request:

Owner: MicrosoftDocs
Repo: PowerShell-Docs
Number: 9065

This report is organized by base folder (a folder containing multiple versions of content) with child folders beneath. Each child folder section enumerates all of the changes to files in that folder across the versions in tables.

Each table includes the version-relative path to the files and the change status of those files for each version. The version-relative path is the remainder of the path from the version folder. For example, the file `reference/5.1/Foo/Bar/Baz.md` is in the `reference` base folder and has a version-relative path of `Foo/Bar/Baz.md`. For change status, every versioned file will have one of the following statuses:

- **Added:** The file was added. Git status `A`.
- **Changed:** The file's type was changed. Git status `T`.
- **Copied:** The file was copied. Git status `C`.
- **Modified:** The file's contents were changed. Git status `M`.
- **Removed:** The file was deleted. Git status `D`.
- **Renamed:** The file was renamed. Git status `R`.
- **Unchanged:** The file was not modified.
- **N/A:** The file does not exist for this version.

In cases where the change status of a file is not the same across versions, the Pull Request author and reviewer should both double check the file to understand if the discrepancy is intentional or accidental.

Base Folder: `reference`

Changed 1 versioned files in `reference`. See below for details per folder.

Change Sets for `Microsoft.PowerShell.LocalAccounts`

| Version-Relative Path | 5.1 | 7.0 | 7.2 | 7.3 |
|-------------------------------|----------|-----|-----|-----|
| <code>Set-LocalUser.md</code> | Modified | N/A | N/A | N/A |

Job summary generated at run-time

Next steps

[PowerShell-Docs style guide](#)

Additional resources

[How we manage pull requests](#)

Contributing quality improvements

Article • 03/30/2025

For [Hacktoberfest 2022](#), we [piloted a process](#) for contributing quality improvements to the PowerShell content. This guide generalizes that process to define a low-friction way for community members to improve to the quality of the documentation.

We're focusing on these quality areas:

- [Formatting code samples](#)
- [Formatting command syntax](#)
- [Link References](#)
- [Markdown linting](#)
- [Spelling](#)

Project Tracking

We're tracking contributions with the [PowerShell Docs Quality Contributions](#) GitHub Project.

The project page has several views for the issues and PRs related to this effort:

- The [Open issues](#) view shows all open issues.
- The [Completed](#) view shows merged PRs.
- The [PowerShell](#) view shows open issues for documentation in the [PowerShell-Docs](#), [PowerShell-Docs-DSC](#), and [PowerShell-Docs-Modules](#) repositories.
- The [Windows PowerShell](#) view shows open issues for documentation in the [windows-powershell-docs repository](#).
- The [Azure PowerShell](#) view shows open issues for documentation in the [azure-docs-powershell repository](#).
- The [Open PRs](#) view shows all open PRs.

Formatting code samples

All code samples should follow the [style guidelines](#) for PowerShell content. Those rules are repeated in abbreviated form here for convenience:

- All code blocks should be contained in a triple-backtick code fence (`````).
- Line length for code blocks is limited to 90 characters for cmdlet reference articles.
- Line length for code blocks is limited to 76 characters for `about_*` articles.
- Avoid using line continuation characters (`) in PowerShell code examples.

- Use splatting or natural line break opportunities, like after pipe (|) characters, opening braces ({), parentheses ((), and brackets ([) to limit line length.
- Only include the PowerShell prompt for illustrative examples where the code isn't intended for copy-pasting.
- Don't use aliases in examples unless you're specifically documenting the alias.
- Avoid using positional parameters. Use the parameter name, even if the parameter is positional.

Formatting command syntax

All prose should follow the [style guidelines](#) for PowerShell content. Those rules are repeated here for convenience:

- Always use the full name for cmdlets and parameters. Avoid using aliases unless you're specifically demonstrating the alias.
- Property, parameter, object, type names, class names, class methods should be **bold**.
 - Property and parameter values should be wrapped in backticks (`).
 - When referring to types using the bracketed style, use backticks. For example:
`[System.IO.FileInfo]`
- PowerShell module names should be **bold**.
- PowerShell keywords and operators should be all lowercase.
- Use proper (Pascal) casing for cmdlet names and parameters.
- When you refer to a parameter by name, the name should be **bold**.
- Use parameter name with the hyphen if you're illustrating syntax. Wrap the parameter in backticks.
- When you show example usage of an external command, the example should be wrapped in backticks. Always include the file extension of the external command.

Link references

For maintainability and readability of the markdown source for our documentation, we're converting our conceptual documentation to use link references instead of inline links.

For example, instead of:

Markdown

The [\[Packages tab\]](#)^[31] displays all available packages in the PowerShell Gallery.

It should be:

Markdown

The [Packages tab][31] displays all available packages in the PowerShell Gallery.

! Note

When you replace an inline link, reflow the content to wrap at 100 characters. You can use the [reflow-markdown](#) VS Code extension to quickly reflow the paragraph.

At the bottom of the file, add a markdown comment before the definition of the links.

Markdown

```
<!-- Link references -->
[01]: https://www.powershellgallery.com/packages
```

Make sure that:

1. The links are defined in the order they appear in the document.
2. Every link points to the correct location.
3. The link reference definitions are at the bottom of the file after the markdown comment and are in the correct order.

Markdown linting

For any article in one of the participating repositories, opening the article in VS Code displays linting warnings. Address any of these warnings you find, except:

- [MD022/blanks-around-headings/blanks-around-headers](#) for the `Synopsis` header in cmdlet reference documents. For those items, the rule violation is intentional to ensure the documentation builds correctly with PlatyPS.

Make sure of the line lengths and use the [Reflow Markdown](#) extension to fix any long lines.

Spelling

Despite our best efforts, typos and misspellings get through and end up in the documentation. These mistakes make documentation harder to follow and localize. Fixing these mistakes makes the documentation more readable, especially for non-English speakers who rely on accurate translations.

Process

We encourage you to choose one or more of the quality areas and an article (or group of articles) to improve. Use the following steps to guide your work:

1. Check the [project ↗](#) for issues filed for this effort to avoid duplicating efforts.
2. Open a new issue in the appropriate repository:
 - Open an issue in [MicrosoftDocs/PowerShell-Docs ↗](#) for PowerShell reference and conceptual content.
 - Open an issue in [MicrosoftDocs/PowerShell-Docs-Dsc ↗](#) for DSC content
 - Open an issue in [MicrosoftDocs/PowerShell-Docs-Modules ↗](#) for Crescendo, PlatyPS, PSScriptAnalyzer, SecretManagement, and SecretStore content.
 - Open an issue in [MicrosoftDocs/azure-docs-powershell ↗](#) for Azure PowerShell content.
 - Open an issue in [MicrosoftDocs/windows-powershell-docs ↗](#) for Windows PowerShell module content.
3. Follow our [contributor's guide](#) to get setup for making your changes.
4. Submit your pull request. Ensure:
 - a. Your PR title has the `Quality:` prefix.
 - b. Your PR body references the issue it resolves as an unordered list item and uses one of the [linking keywords ↗](#).

For example, if you're working on issue `123`, the body of your PR should include the following Markdown:

Markdown

```
- resolves #123
```

After you submit the PR, the maintainers will review your work and work with you to get it merged.

Hacktoberfest and other hack-a-thon events

Article • 05/29/2025

Hacktoberfest is an annual worldwide event held during October. The event encourages open source developers to contribute to repositories through pull requests (PR). GitHub hosts many open source repositories that contribute to Microsoft Learn content. Several repositories actively participate in Hacktoberfest.

How to contribute

Before you can contribute to an open source repo, you must first configure your account to contribute to Microsoft Learn. If you're new to this process, start by signing up for a [GitHub account](#). Be sure to [install Git and the Markdown tools](#).

To get credit for participation, register with [Hacktoberfest](#) and read their participation guide.

Find a repo that needs your help

The PowerShell-Docs team is supporting Hacktoberfest contributions for several PowerShell documentation repositories. We defined a set of cleanup tasks designed to be simple for first time contributors. Full information can be found in the [Hacktoberfest meta-issue](#).

To be successful with these tasks, you should:

- Have a general understanding of PowerShell syntax
- Have an understanding of [splatting](#)
- Be able to read and follow the [PowerShell-Docs style guide](#) and [Editorial checklist](#)
- Have basic familiarity with Markdown

Before contributing should read the meta-issue. When you're ready to start, open a new Hacktoberfest using one of the following links:

- [MicrosoftDocs/PowerShell-Docs](#)
- [MicrosoftDocs/PowerShell-Docs-DSC](#)
- [MicrosoftDocs/PowerShell-Docs-Modules](#)
- [MicrosoftDocs/windows-powershell-docs](#)
- [MicrosoftDocs/azure-docs-powershell](#)

Quality expectations

To have a successful contribution to an open source Microsoft Learn repository, create a meaningful and impactful PR. The following examples from the official Hacktoberfest site are considered ***low-quality contributions***:

- PRs containing bulk automated changes
 - Example: scripted PRs to remove whitespace, fix common spelling, or optimize images
 - Submit an issue first describing the automated changes you want to make
- PRs deemed disruptive (for example, taking someone else's branch or commits and making a PR)
- PRs deemed a hindrance vs. helping
- PRs that are clearly an attempt to increment your PR count for October

Open a PR

A *PR* provides a convenient way for a contributor to propose a set of changes. Successful PRs have these common characteristics:

- The PR adds value.
- The contributor is receptive to feedback.
- The intended changes are well articulated.
- The changes are related to an existing issue.

If you're proposing a PR without a corresponding issue, create an issue first. For more information, see [GitHub: About pull requests ↗](#).

See also

- [Git and GitHub essentials for Microsoft Learn documentation](#)
- [Official Hacktoberfest site ↗](#)

How we manage issues

Article • 03/30/2025

This article documents how we manage issues in the PowerShell-Docs repository. This article is designed to be a job aid for members of the PowerShell-Docs team. We publish this information here to provide process transparency for our public contributors.

Sources of issues

- Community contributors
- Internal contributors
- Transcriptions of comments from social media channels
- Feedback via the Docs feedback form

Response time targets

80% of new issues are closed within 3 business days.

- Triaged - 1 business day
- Fix or change - 10 business days

Labeling & Milestones

Label Types

- **Area** - Identifies the part of PowerShell or the docs that the issue is discussing
- **Issue** - The type of issue: like bug, feedback, or idea
- **Priority** - The priority of the issue; value range 0-3 (high-low)
- **Quality** - The [quality improvement](#) effort the issue commits to resolving
- **Status** - The status of the work item or why it was closed
- **Tag** - Used to for additional classification like availability or doc-a-thons
- **Waiting** - Shows that we're waiting on some external person or event

For more information on specific labels, see [Labeling](#).

Milestones

Issues and PRs should be tagged with the appropriate milestone. If the issue doesn't apply to a specific version, then no milestone is used. PRs and related issues for changes that have yet to

be merged into the PowerShell code base should be assigned to the **Future** milestone. After you merge the change, update the milestone to the appropriate version.

[] [Expand table](#)

| Milestone | Description |
|-----------|---|
| 7.0.0 | Work items related to PowerShell 7.0 |
| 7.2.0 | Work items related to PowerShell 7.2 |
| 7.3.0 | Work items related to PowerShell 7.3 |
| Future | Work items a future version of PowerShell |

Triage process

PowerShell docs team members review the issues daily and triage new issues as they arrive. The team meets weekly to discuss difficult issues need triage and prioritize the work.

Misplaced product feedback

- Enter a comment redirecting the customer to the correct feedback channel.
- Optional: Copy the issue to the appropriate product feedback location, add a link to the copied item, and close the issue.

The default location for PowerShell issues is

<https://github.com/PowerShell/PowerShell/issues/new/choose>.

Support requests

- If the support question is simple, answer it politely and close the issue.
- If the question is more complicated, or the submitter replies with more questions, redirect them to forums and support channels. Suggested text for redirecting to forums:

Markdown

```
> This is not the right forum for these kinds of questions. Try posting your  
question in a  
> community support forum. For a list of community forums see:  
> https://learn.microsoft.com/powershell/scripting/community/community-support
```

Code of conduct violations

- Edit the issue to remove any offensive content, if necessary
- Enter a comment indicating the issue is spam, close the issue, and then lock it to prevent further comments
- Discuss each violation in the regular triage meeting to determine the need for further action

Managing pull requests

Article • 03/30/2025

This article documents how we manage pull requests in the PowerShell-Docs repository. This article is designed to be a job aid for members of the PowerShell-Docs team. We publish this information here to provide process transparency for our public contributors.

Best practices

- Request a review. The person submitting the PR shouldn't merge the PR without a peer review.
- Assign the peer reviewer when the PR is submitted. Early assignment allows the reviewer to respond sooner with editorial remarks.
- Use comments to describe the nature of the change being submitted. For example, if the change is minor, explain the change and that you don't need a full technical review. Be sure to @mention the reviewer.
- Use the comment suggestion feature to make it easier for the author to accept the suggested change. For more information, see [Reviewing proposed changes in a pull request ↗](#).

PR Process steps

1. Writer: Create PR

- Fill out the [PR template](#)
- Link any issues resolved by the PR
- Use GitHub's [autoclose ↗](#) feature to close the issue
- Work through and check off each item in the checklist

2. Writer: Assign peer reviewer

3. Reviewer: proofreads and comments (as necessary)

4. Writer: Incorporate review feedback

5. Both: Review preview rendering

6. Both: Review validation report - fix warnings and errors

7. Reviewer: Mark review "Approved"

8. Repo Maintainer: Merge PR

Content Reviewer Checklist

See the [editorial checklist](#) for a more comprehensive list.

- Proofread for grammar, style, concision, technical accuracy
- Ensure examples still apply for the target version
- Check Preview rendering
- Check metadata - ms.date, remove ms.assetid, ensure required fields
- Validate markdown correctness
 - See style guide for content-specific formatting rules
- Reorganize examples as follows:
 - Intro paragraph
 - Code and output
 - Detailed explanation of code (as necessary)
- Check hyperlinks for accuracy
 - Replace or remove TechNet/MSDN links
 - Ensure minimum number of redirects to target
 - Ensure HTTPS
 - Correct link type
 - File links for local files
 - URL links for files outside of the docset
 - Remove locales from URLs
 - Simplify URLs pointing to `learn.microsoft.com`
- Verify versioned content is correct across all versions
 - Review the [versioned content change report](#)

Branch Merge Process

The `main` branch is the only branch that should be merged into `live`. Merges from short-lived (working) branches should be squashed before merging into `main`.

[] [Expand table](#)

| Merge from/to | <i>release-branch</i> | <i>main</i> | <i>live</i> |
|-----------------------------|------------------------------|--------------------|--------------------|
| <code>working-branch</code> | squash and merge | squash and merge | Not allowed |
| <code>release-branch</code> | — | merge | Not allowed |
| <code>main</code> | rebase | — | merge |

PR Merger checklist

- Content review complete
- Correct target branch for the change

- No merge conflicts
- All validation and build step pass
 - Warnings and suggestions should be fixed (see [Notes](#) for exceptions)
 - No broken links
 - The [Checklist](#) action ran and passed
 - If an [Authorization](#) check was triggered, it passed
- Merge according to table

Notes

The following warnings can be ignored:

Can't find service name for `<version>/<modulepath>/About/About.md`

Metadata with following name(s) are not allowed to be set in YAML header, or as file level metadata in docfx.json, or as global metadata in docfx.json: `locale`. They are generated by Docs platform, so the values set in these 3 places will be ignored. Please remove them from all 3 places to resolve the warning.

When a PR is merged, the HEAD of the target branch is changed. Any open PRs that were based on the previous HEAD are now outdated. A Maintainer has the right required to override the merge warnings and merge the outdated PR in GitHub. Merging an outdated PR is safe if the previously merged PRs didn't touch the same files.

To update the PR, select the **Update Branch** button. Choose **Update with rebase** option. For more information, see [Updating your pull request branch ↗](#).

Publishing to Live

Periodically, the changes accumulated in the `main` branch need to be published to the live website.

- The `main` branch is merged to `live` each weekday at 3pm PST.
- The `main` branch should be merged to `live` after any significant change.
 - Changes to 50 or more files
 - After merging a release branch

- Changes to repo or docset configurations (docfx.json, OPS configs, build scripts, etc.)
- Changes to the redirection file
- Changes to the TOC
- After merging a "project" branch (content reorg, bulk update, etc.)

Labeling in GitHub

Article • 03/30/2025

This article documents how we label issues and pull requests in the PowerShell-Docs repository. This article is designed to be a job aid for members of the PowerShell-Docs team. We publish this information here to provide process transparency for our public contributors.

Labels always have a name and a description that is prefixed with their type.

Area labels

Area labels identify the parts of PowerShell or the documentation that the issue relates to.

[+] Expand table

| Label | Related Content |
|---------------------|---|
| area-about | The <code>about_*</code> articles. |
| area-archive | The Microsoft.PowerShellArchive module. |
| area-cim | The CimCmdlets module. |
| area-community | Community-facing projects, including the contributor's guide and monthly updates. |
| area-conceptual | Conceptual articles (not cmdlet reference). |
| area-console | The console host |
| area-core | The Microsoft.PowerShellCore module. |
| area-crescendo | The Crescendo module. |
| area-debugging | Debugging PowerShell. |
| area-diagnostics | The Microsoft.PowerShellDiagnostics module. |
| area-dsc | PowerShell Desired State Configuration. |
| area-editorsvcs | The PowerShell editor services. |
| area-engine | The PowerShell engine. |
| area-error-handling | Error handling in PowerShell |

| Label | Related Content |
|-------------------------------------|---|
| <code>area-experimental</code> | PowerShell's experimental features |
| <code>area-gallery</code> | The PowerShell Gallery. |
| <code>area-helpsystem</code> | The Help services, including the pipeline and <code>*-Help</code> cmdlets. |
| <code>area-host</code> | The Microsoft.PowerShell.Host module. |
| <code>area-ise</code> | The PowerShell ISE. |
| <code>area-jea</code> | The Just Enough Administration feature. |
| <code>area-language</code> | The PowerShell syntax and keywords. |
| <code>area-learn</code> | The structured training content for PowerShell. |
| <code>area-localaccounts</code> | The Microsoft.PowerShell.LocalAccounts module. |
| <code>area-localization</code> | Localization problems or opportunities for the content. |
| <code>area-management</code> | The Microsoft.PowerShell.Management module. |
| <code>area-native-cmds</code> | Using native commands in PowerShell. |
| <code>area-omi</code> | Open Management Infrastructure & CDXML. |
| <code>area-ops-issue</code> | Building and rendering the content on the site. |
| <code>area-other</code> | Miscellaneous modules. |
| <code>area-overview</code> | The overview section in the conceptual content. |
| <code>area-packageManagement</code> | The PackageManagement module. |
| <code>area-parallelism</code> | Content covering parallel processing, such as using <code>ForEach-Object</code> or PowerShell Jobs. |
| <code>area-platypus</code> | The Platypus module. |
| <code>area-portability</code> | Cross-platform compatibility. |
| <code>area-powershellget</code> | The PowerShellGet module. |
| <code>area-providers</code> | PowerShell providers. |
| <code>area-psreadline</code> | The PSReadLine module. |
| <code>area-release-notes</code> | The PowerShell release notes. |

| Label | Related Content |
|---------------------|--|
| area-remoting | The PowerShell remoting feature and cmdlets. |
| area-scriptanalyzer | The PSScriptAnalyzer module. |
| area-sdk-docs | The conceptual documentation for the PowerShell SDK. |
| area-sdk-ref | The .NET API reference documentation for the PowerShell SDK. |
| area-security | The Microsoft.PowerShell.Security module and security concepts in general. |
| area-setup | Installing and configuring PowerShell. |
| area-threadjob | The ThreadJob module. |
| area-utility | The Microsoft.PowerShell.Utility module. |
| area-versions | Issues with the versioning of the documentation. |
| area-vscode | The VS Code PowerShell extension. |
| area-wincompat | The Windows Compatibility feature. |
| area-wmf | The Windows Management Framework. |
| area-workflow | The Windows PowerShell Workflow feature. |

Issue labels

Issue labels distinguish issues by purpose.

[\[\] Expand table](#)

| Label | Issue Category |
|------------------------|---|
| issue-doc-bug | Errors or ambiguities in the content |
| issue-doc-idea | Requests for new content |
| issue-kudos | Praise, positive feedback, or thanks rather than work items |
| issue-product-feedback | Feedback or problems with the product itself |
| issue-question | Support questions |

Priority labels

Priority labels rank which work items need to be worked on before others. These labels are only used when needed to manage large sets of work items.

[\[\] Expand table](#)

| Label | Priority Level |
|-------|----------------|
| Pri0 | Highest |
| Pri1 | High |
| Pri2 | Medium |
| Pri3 | Low |

Project Labels

Project labels indicate what ongoing GitHub Project a work item is related to. These labels are used for automatically adding work items to a project on creation.

[\[\] Expand table](#)

| Label | Project |
|-----------------|---|
| project-quality | The quality improvement project |

Quality labels

Quality labels categorize work items for the [quality improvement](#) effort.

[\[\] Expand table](#)

| Label | Improvement |
|-------------------------------|---|
| quality-aliases | Ensure cmdlet aliases are documented |
| quality-format-code-samples | Ensure proper casing, line length, and other formatting in code samples |
| quality-format-command-syntax | Ensure proper casing and formatting for command syntax |
| quality-link-references | Ensure links in conceptual docs are defined as numbered references |

| Label | Improvement |
|----------------------|---|
| quality-markdownlint | Ensure content follows markdownlint rules |
| quality-spelling | Ensure proper casing and spelling for words |

Status labels

Status labels indicate why a work item was closed or shouldn't be merged. Issues are only given status labels when they're closed without a related PR.

[\[+\] Expand table](#)

| Label | Status |
|-----------------------------|---|
| resolution-answered | Closed by existing documentation |
| resolution-duplicate | Closed as duplicate issue |
| resolution-external | Closed by customer or outside resource |
| resolution-no-repro | Unable to reproduce the reported issue |
| resolution-refer-to-support | Closed and referred to community or product support |
| resolution-wont-fix | Closed as won't fix |

Tag labels

Tag labels add independent context for work items.

[\[+\] Expand table](#)

| Label | Purpose |
|-------------------------|---|
| in-progress | Someone is actively working on the item |
| go-live | The work item is related to a specific release |
| doc-a-thon | The work item is related to a doc-a-thon |
| up-for-grabs | Any contributor can volunteer to resolve the work item |
| hacktoberfest-accepted | The PR is accepted for inclusion in #hacktoberfest |
| hacktoberfest-candidate | The PR is a candidate for inclusion in #hacktoberfest |

| Label | Purpose |
|-----------------|--|
| needs-triage | The issue must be triaged by the team before it's ready to be worked |
| code-of-conduct | Closed for spam, trolling, or code of conduct violations |
| do-not-merge | The PR isn't meant to be merged |

Waiting labels

Waiting labels indicate that a work item can't be resolved until an external condition is met.

[\[\] Expand table](#)

| Label | Waiting For |
|---------------------|---|
| hold-for-pr | Upstream PR to be merged |
| hold-for-release | Upstream product to release |
| needs-investigation | Waiting for team member to verify or research |
| needs-more-info | Additional details or clarification from work item author |
| needs-response | Response from work item author |
| review-shiproom | Shiproom discussion with the PowerShell team |

PowerShell Support Lifecycle

Article • 02/25/2025

ⓘ Note

This document is about support for PowerShell. Windows PowerShell (1.0 - 5.1) is a component of the Windows operating system. For more information, see [Product and Services Lifecycle Information](#).

PowerShell follows the [Microsoft Modern Lifecycle Policy](#). Support dates follow the [.NET Support Policy](#). In this servicing approach, customers can choose Long Term Support (LTS) releases or current releases.

An **LTS** release of PowerShell is built on an **LTS** release of .NET. Updates to an **LTS** release only contain critical security updates and servicing fixes that are designed to minimize impact on existing workloads.

A **current** release is a release that occurs between **LTS** releases. Current releases can contain critical fixes, innovations, and new features. Microsoft supports a **current** release for six months after the next **LTS** release.

Both **LTS** and **current** versions of PowerShell receive security updates and bug fixes. Microsoft only supports the latest update version of a release.

Getting support

Microsoft provides support for PowerShell on a best-effort basis. Support for Windows PowerShell 5.1 is provided through Windows support channels. You can use the standard paid support channels to get support for PowerShell.

- [Support for business](#)
- [Contact support](#)

There are many free support options available from the PowerShell community. The most active community support channels are available through **Discord** or **Slack**. The discussion channels are mirrored on both platforms, so you can choose the platform that you prefer. These channels can help you troubleshoot issues, answer questions, and provide guidance on how to use PowerShell.

If you think that you found a bug, you can file an issue on [GitHub](#). The PowerShell team can't provide support through GitHub, but they welcome bug reports. The

[community support page](#) provides links to the most popular community support channels.

Supported platforms

PowerShell runs on multiple operating systems (OS) and processor architecture platforms. The platform must meet the following criteria:

- The target platform (OS version and processor architecture) is supported by .NET.
- Microsoft has tested and approved PowerShell on the target platform.
- The OS version is supported by the distributor for at least one year.
- The OS version isn't an interim release or equivalent.
- The OS version is currently supported by the distributor.

Support for PowerShell ends when either of the following conditions are met:

- The target platform reaches end-of-life as defined by the platform owner
- The specific version of PowerShell reaches end-of-life

After a version of PowerShell reaches end-of-life, no further updates, including security updates, are provided. Microsoft encourages customers to upgrade to a supported version of PowerShell to continue receiving updates and support.

Windows

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [Windows reaches end-of-support](#).

- Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 for Windows Server 2022, Windows Server Core 2022, and Windows Server Nano build 1809 are available from the [Microsoft Artifact Registry](#).
- PowerShell 7.4 and higher can be installed on Windows 10 build 1607 and higher, Windows 11, Windows Server 2016 and higher.

Note

Support for a specific version of Windows is determined by the Microsoft Support Lifecycle policies. For more information, see:

- [Windows client lifecycle FAQ](#)
- [Modern Lifecycle Policy FAQ](#)

macOS

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of macOS reaches end-of-support.

- macOS 15 (Sequoia) x64 and Arm64
- macOS 14 (Sonoma) x64 and Arm64
- macOS 13 (Ventura) x64 and Arm64

Apple determines the support lifecycle of macOS. For more information, see the following:

- [macOS release notes ↗](#)
- [Apple Security Updates ↗](#)

Alpine Linux

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [Alpine reaches end-of-life ↗](#).

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 are available from the [Microsoft Artifact Registry ↗](#) for the following versions of Alpine:

- Alpine 3.20 - OS support ends on 2026-04-01

Docker images of PowerShell aren't available for Alpine 3.21.

Important

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates. Microsoft recommends that you update the OS packages to the latest version to ensure the latest security updates are applied.

Debian Linux

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [Debian reaches end-of-life ↗](#).

Install package files (.deb) are also available from <https://packages.microsoft.com/> ↗.

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 are available from the [Microsoft Artifact Registry ↗](#) for the following versions of Debian:

- Debian 12 (Bookworm) - OS support ends on 2026-06-10

Important

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates.

Microsoft recommends that you update the OS packages to the latest version to ensure the latest security updates are applied.

Red Hat Enterprise Linux (RHEL)

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [RHEL reaches end-of-support](#).

Install package files (.rpm) are also available from <https://packages.microsoft.com/>.

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 are available from the [Microsoft Artifact Registry](#) for the following versions of RHEL:

- RHEL 9 - OS support ends on 2032-05-31
- RHEL 8 - OS support ends on 2029-05-31

PowerShell is tested on Red Hat Universal Base Images (UBI). For more information, see the [UBI information page](#).

Important

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates.

Microsoft recommends that you update the OS packages to the latest version to ensure the latest security updates are applied.

Ubuntu Linux

Microsoft supports PowerShell until [PowerShell reaches end-of-support](#) or the version of [Ubuntu reaches end-of-support](#).

Install package files (.deb) are also available from <https://packages.microsoft.com/>.

Docker images containing PowerShell 7.4 and PowerShell 7.5-preview for x64 and Arm32 are available from the [Microsoft Artifact Registry](#) for the following versions of

Ubuntu:

- Ubuntu 24.04 (Noble Numbat) - OS support ends on 2029-04-01
- Ubuntu 22.04 (Jammy Jellyfish) - OS support ends on 2027-04-01
- Ubuntu 20.04 (Focal Fossa) - OS support ends on 2025-04-02

Ubuntu 24.10 (Oracular Oriole) is an interim release. Microsoft doesn't support [interim releases](#) of Ubuntu. For more information, see [Community supported distributions](#).

Important

The Docker images are built from official operating system (OS) images provided by the OS distributor. These images may not have the latest security updates. Microsoft recommends that you update the OS packages to the latest version to ensure the latest security updates are applied.

Support for PowerShell modules

The support lifecycle for PowerShell doesn't cover modules that ship outside of the PowerShell release package. For example, using the `ActiveDirectory` module that ships as part of Windows Server is supported under the [Windows Support Lifecycle](#).

Support for experimental features

[Experimental features](#) aren't intended to be used in production environments. We appreciate feedback on experimental features and we provide best-effort support for them.

Notes on licensing

PowerShell is released under the [MIT license](#). Under this license, and without a paid support agreement, users are limited to [community support](#). With community support, Microsoft makes no guarantees of responsiveness or fixes.

PowerShell end-of-support dates

The PowerShell support lifecycle follows the [support lifecycle of .NET](#). The following table lists the end-of-support dates for the current versions of PowerShell:

[\[+\] Expand table](#)

| Version | Release Date | End-of-support | .NET Version |
|--------------------------|--------------|----------------|---------------------------------------|
| PowerShell 7.6 (preview) | Future date | Future date | Built on .NET 9.0.0 ↗ |
| PowerShell 7.5 | 23-Jan-2025 | 12-May-2026 | Built on .NET 9.0.0 ↗ |
| PowerShell 7.4 (LTS) | 16-Nov-2023 | 10-Nov-2026 | Built on .NET 8.0.0 ↗ |

The following table lists the end-of-support dates for retired versions of PowerShell:

[\[+\] Expand table](#)

| Version | Release Date | End-of-support | .NET Version |
|----------------------|--------------|----------------|--|
| PowerShell 7.3 | 09-Nov-2022 | 08-May-2024 | Built on .NET 7.0 ↗ |
| PowerShell 7.2 (LTS) | 08-Nov-2021 | 08-Nov-2024 | Built on .NET 6.0 ↗ |
| PowerShell 7.1 | 11-Nov-2020 | 08-May-2022 | Built on .NET 5.0 ↗ |
| PowerShell 7.0 (LTS) | 04-Mar-2020 | 03-Dec-2022 | Built on .NET Core 3.1 ↗ |
| PowerShell 6.2 | 29-Mar-2019 | 04-Sep-2020 | Built on .NET Core 2.1 ↗ |
| PowerShell 6.1 | 13-Sep-2018 | 28-Sep-2019 | Built on .NET Core 2.1 ↗ |
| PowerShell 6.0 | 20-Jan-2018 | 13-Feb-2019 | Built on .NET Core 2.0 ↗ |

Windows PowerShell release history

The following table contains a historical timeline of the major releases of Windows PowerShell. Microsoft no longer supports Windows PowerShell versions lower than 5.1.

[\[+\] Expand table](#)

| Version | Release Date | Note |
|------------------------|--------------|--|
| Windows PowerShell 5.1 | Aug-2016 | Released in Windows 10 Anniversary Update and Windows Server 2016, WMF 5.1 |
| Windows PowerShell 5.0 | Feb-2016 | Released in Windows Management Framework (WMF) 5.0 |
| Windows PowerShell 4.0 | Oct-2013 | Released in Windows 8.1 and with Windows Server 2012 R2, WMF 4.0 |

| Version | Release Date | Note |
|------------------------|---------------------|--|
| Windows PowerShell 3.0 | Oct-2012 | Released in Windows 8 and with Windows Server 2012 WMF 3.0 |
| Windows PowerShell 2.0 | Jul-2009 | Released in Windows 7 and Windows Server 2008 R2, WMF 2.0 |
| Windows PowerShell 1.0 | Nov-2006 | Released as optional component of Windows Server 2008 |

Run the following command to see the full version number of .NET used by the version of PowerShell you're running:

```
PowerShell
[System.Runtime.InteropServices.RuntimeInformation]::FrameworkDescription
```