about_Automatic_Variables - PowerShell

sdwheeler ⋮ 30-38 minutes

about_Automatic_Variables

- Article
- 07/12/2024
-

In this article

Short description

Describes variables that store state information for and are created and maintained by PowerShell.

Conceptually, most of these variables are considered to be read-only. Even though they *can* be written to, for backward compatibility they *should not* be written to.

Here is a list of the automatic variables in PowerShell:

- $
- $?
- $^
- $_
- $args
- $ConsoleFileName
- $EnabledExperimentalFeatures
- $Error
- $Event
- $EventArgs
- $EventSubscriber
- $ExecutionContext
- $false
- $foreach
- $HOME
- $Host

- $input
- $IsCoreCLR
- $IsLinux
- $IsMacOS
- $IsWindows
- $LASTEXITCODE
- $Matches
- $MyInvocation
- $NestedPromptLevel
- $null
- $PID
- $PROFILE
- $PSBoundParameters
- $PSCmdlet
- $PSCommandPath
- $PSCulture
- $PSDebugContext
- $PSEdition
- $PSHOME
- $PSItem
- $PSScriptRoot
- $PSSenderInfo
- $PSUICulture
- $PSVersionTable
- $PWD
- $Sender
- $ShellId
- $StackTrace
- $switch
- $this
- $true

Long description

$

Contains the last token in the last line received by the session.

$?

Contains the execution status of the last command. It contains **True** if the last command succeeded and **False** if it failed. Parse errors don't result in execution, so they don't affect the value of `$?`.

For cmdlets and advanced functions that are run at multiple stages in a pipeline, for example in both `process` and `end` blocks, calling `this.WriteError()` or `$PSCmdlet.WriteError()` respectively at any point sets `$?` to **False**, as does `this.ThrowTerminatingError()` and `$PSCmdlet.ThrowTerminatingError()`.

The `Write-Error` cmdlet always sets `$?` to **False** immediately after it's executed, but won't set `$?` to **False** for a function calling it:

```
function Test-WriteError
{
    Write-Error "Bad"
    "The `$? variable is: $?"
}

Test-WriteError
"Now the `$? variable is: $?"
```

```
Test-WriteError:
Line |
   7 |  Test-WriteError
     |  ~~~~~~~~~~~~~~~
     | Bad
The $? variable is: False
Now the $? variable is: True
```

For the latter purpose, `$PSCmdlet.WriteError()` should be used instead.

For native commands (executables), `$?` is set to **True** when `$LASTEXITCODE` is `0`, and set to **False** when `$LASTEXITCODE` is any other value.

Note

Until PowerShell 7, wrapping a statement within parentheses `(...)`, subexpression syntax `$(...)`, or an array expression `@(...)` always reset `$?` to **True**. For example, `(Write-Error)` shows `$?` as **True**. This behavior changed in PowerShell 7, so that `$?` always reflects the actual success of the last command run in these expressions.

$^

Contains the first token in the last line received by the session.

$_

Same as `$PSItem`. Contains the current object in the pipeline object. You can use this variable in commands that perform an action on every object in a pipeline.

For more information, see about_PSItem.

$args

Contains an array of values for undeclared parameters that are passed to a function, script, or script block. When you create a function, you can declare the parameters with the `param` keyword or by adding a comma-separated list of parameters in parentheses after the function name.

In an event action, the `$args` variable contains objects that represent the event arguments of the event that's being processed. This variable is populated only within the `Action` block of an event registration command. The value of this variable can also be found in the **SourceArgs** property of the **PSEventArgs** object that `Get-Event` returns.

$ConsoleFileName

Contains the path of the console file (`.psc1`) that was most recently used in the session. This variable is populated when you start PowerShell with the **PSConsoleFile** parameter or when you use the

`Export-Console` cmdlet to export snap-in names to a console file.

When you use the `Export-Console` cmdlet without parameters, it automatically updates the console file that was most recently used in the session. You can use this automatic variable to determine the file to update.

$EnabledExperimentalFeatures

Contains a list of names of the experimental features that are enabled.

$Error

Contains an array of error objects that represent the most recent errors. The most recent error is the first error object in the array `$Error[0]`.

To prevent an error from being added to the `$Error` array, use the **ErrorAction** common parameter with a value of **Ignore**. For more information, see about_CommonParameters.

$Event

Contains a **PSEventArgs** object that represents the event that's being processed. This variable is populated only within the `Action` block of an event registration command, such as `Register-ObjectEvent`. The value of this variable is the same object that the `Get-Event` cmdlet returns. You can use the properties of the `Event` variable, such as `$Event.TimeGenerated`, in an `Action` script block.

$EventArgs

Contains an object that represents the first event argument that derives from **EventArgs** of the event that's being processed. This variable is populated only within the `Action` block of an event registration command. The value of this variable can also be found in the **SourceEventArgs** property of the **PSEventArgs** object that `Get-Event` returns.

$EventSubscriber

Contains a **PSEventSubscriber** object that represents the event subscriber of the event that's being processed. This variable is populated only within the `Action` block of an event registration command. The value of this variable is the same object that the `Get-EventSubscriber` cmdlet returns.

$ExecutionContext

Contains an **EngineIntrinsics** object that represents the execution context of the PowerShell host. You can use this variable to find the execution objects that are available to cmdlets.

$false

Contains **False**. You can use this variable to represent **False** in commands and scripts instead of using the string `"false"`. The string can be interpreted as **True** if it's converted to a non-empty string or to a non-zero integer.

$foreach

Contains the enumerator (not the resulting values) of a ForEach loop. The `$ForEach` variable exists only while the `ForEach` loop is running; it's deleted after the loop is completed.

Enumerators contain properties and methods you can use to retrieve loop values and change the current loop iteration. For more information, see Using Enumerators.

$HOME

Contains the full path of the user's home directory. On Windows, this variable uses the value of the `"$env:USERPROFILE"` Windows environment variable, typically `C:\Users\<UserName>`. On Unix, this variable uses the value of the `HOME` environment variable.

Important

Windows can redirect the location of the user's profile. This means that `$HOME` may not have the same value as `"$env:HOMEDRIVE$env:HOMEPATH"`.

$Host

Contains an object that represents the current host application for PowerShell. You can use this variable to represent the current host in commands or to display or change the properties of the host, such as `$Host.version` or `$Host.CurrentCulture`, or `$Host.UI.RawUI.BackGroundColor = "Red"`.

Note

The color settings in `$Host.PrivateData` have been replaced by the `$PSStyle` preference variable. For more information, see about_ANSI_Terminals.

$input

Contains an enumerator that enumerates all input that's passed to a function. The `$input` variable is available only to functions, script blocks (which are unnamed functions), and script files (which are saved script blocks).

- In a function without a `begin`, `process`, or `end` block, the `$input` variable enumerates the collection of all input to the function.
- In the `begin` block, the `$input` variable contains no data.
- In the `process` block, the `$input` variable contains the current object in the pipeline.
- In the `end` block, the `$input` variable enumerates the collection of all input to the function.

  Note

  You can't use the `$input` variable inside both the `process` block and the `end` block in the same function or script block.

Since `$input` is an enumerator, accessing any of its properties causes `$input` to no longer be available. You can store `$input` in another variable to reuse the `$input` properties.

Enumerators contain properties and methods you can use to retrieve loop values and change the current loop iteration. For more information, see Using Enumerators.

The `$input` variable is also available to the command specified by the `-Command` parameter of `pwsh` when invoked from the command line. The following example is run from the Windows Command shell.

```
echo Hello | pwsh -Command """$input World!"""
```

$IsCoreCLR

Contains `$true` if the current session is running on the .NET Core Runtime (CoreCLR). Otherwise contains `$false`.

$IsLinux

Contains `$true` if the current session is running on a Linux operating system. Otherwise contains `$false`.

$IsMacOS

Contains `$true` if the current session is running on a MacOS operating system. Otherwise contains `$false`.

$IsWindows

Contains `$true` if the current session is running on a Windows operating system. Otherwise contains `$false`.

$LASTEXITCODE

Contains the exit code of the last native program or PowerShell script that ran.

For PowerShell scripts, the value of `$LASTEXITCODE` depends on how the script was called and whether the `exit` keyword was used:

- When a script uses the `exit` keyword:

  `$LASTEXITCODE` is set to value the specified by the `exit` keyword. For more information, see about_Language_Keywords.

- When a script is called directly, like `./Test.ps1`, or with the call operator (&) like `& ./Test.ps1`:

  The value of `$LASTEXITCODE` isn't changed unless:

  - The script calls another script that uses the `exit` keyword
  - The script calls a native command
  - The script uses the `exit` keyword

- When a script is called with `pwsh` using the **File** parameter, `$LASTEXITCODE` is set to:

  - `1` if the script terminated due to an exception
  - The value specified by the `exit` keyword, if used in the script
  - `0` if the script completed successfully

- When a script is called with `pwsh` using the **Command** parameter, `$LASTEXITCODE` is set to:

  - 1 if the script terminated due to an exception or if the result of the last command set `$?` to `$false`
  - `0` if the script completed successfully and the result of the last command set `$?` to `$true`

For more information on the **File** and **Command** parameters, see about_Pwsh.

$Matches

The `$Matches` variable works with the `-match` and `-notmatch` operators. When you submit scalar input to the `-match` or `-notmatch` operator, and either one detects a match, they return a Boolean value and populate the `$Matches` automatic variable with a hash table of any string values that were matched. The `$Matches` hash table can also be populated with captures when you use regular expressions with the `-match` operator.

For more information about the `-match` operator, see about_Comparison_Operators. For more information on regular expressions, see about_Regular_Expressions.

The `$Matches` variable also works in a `switch` statement with the `-Regex` parameter. It's populated the same way as the `-match` and `-notmatch` operators. For more information about the `switch` statement, see about_Switch.

Note

When `$Matches` is populated in a session, it retains the matched value until it's overwritten by another match. If `-match` is used again and no match is found, it doesn't reset `$Matches` to `$null`. The previously matched value is kept in `$Matches` until another match is found.

$MyInvocation

Contains information about the current command, such as the name, parameters, parameter values, and information about how the command was started, called, or invoked, such as the name of the script that called the current command.

`$MyInvocation` is populated only for scripts, function, and script blocks. You can use the information in the **System.Management.Automation.InvocationInfo** object that `$MyInvocation` returns in the current script, such as the name of a function (`$MyInvocation.MyCommand.Name`) to identify the current command. This is useful for finding the name of the current script.

Beginning in PowerShell 3.0, `MyInvocation` has the following new properties.

- **PSScriptRoot** - Contains the full path to the script that invoked the current command. The value of this property is populated only when the caller is a script.
- **PSCommandPath** - Contains the full path and filename of the script that invoked the current command. The value of this property is populated only when the caller is a script.

Unlike the `$PSScriptRoot` and `$PSCommandPath` automatic variables, the **PSScriptRoot** and **PSCommandPath** properties of the `$MyInvocation` automatic variable contain information about the invoker or calling script, not the current script.

$NestedPromptLevel

Contains the current prompt level. A value of 0 indicates the original prompt level. The value is incremented when you enter a nested level and decremented when you exit it.

For example, PowerShell presents a nested command prompt when you use the `$Host.EnterNestedPrompt` method. PowerShell also presents a nested command prompt when you reach a breakpoint in the PowerShell debugger.

When you enter a nested prompt, PowerShell pauses the current command, saves the execution context, and increments the value of the `$NestedPromptLevel` variable. To create additional nested command prompts (up to 128 levels) or to return to the original command prompt, complete the command, or type `exit`.

The `$NestedPromptLevel` variable helps you track the prompt level. You can create an alternative PowerShell command prompt that includes this value so that it's always visible.

$null

`$null` is an automatic variable that contains a **null** or empty value. You can use this variable to represent an absent or undefined value in commands and scripts.

PowerShell treats `$null` as an object with a value, or a placeholder, so you can use `$null` to represent an empty value in a collection of values.

For example, when `$null` is included in a collection, it's counted as one of the objects.

```
$a = "one", $null, "three"
$a.count
```

```
3
```

If you pipe the `$null` variable to the `ForEach-Object` cmdlet, it generates a value for `$null`, just as it does for the other objects

```
"one", $null, "three" | ForEach-Object { "Hello " + $_}
```

```
Hello one
Hello
Hello three
```

As a result, you can't use `$null` to mean **no parameter value**. A parameter value of `$null` overrides the default parameter value.

However, because PowerShell treats the `$null` variable as a placeholder, you can use it in scripts like the following one, which wouldn't work if `$null` were ignored.

```
$calendar = @($null, $null, "Meeting", $null, $null, "Team Lunch",
$null)
$days = "Sunday","Monday","Tuesday","Wednesday","Thursday",
        "Friday","Saturday"
$currentDay = 0
```

```
foreach($day in $calendar)
{
    if($day -ne $null)
    {
        "Appointment on $($days[$currentDay]): $day"
    }

    $currentDay++
}
```

```
Appointment on Tuesday: Meeting
Appointment on Friday: Team lunch
```

$PID

Contains the process identifier (PID) of the process that's hosting the current PowerShell session.

$PROFILE

Contains the full path of the PowerShell profile for the current user and the current host application. You can use this variable to represent the profile in commands. For example, you can use it in a command to determine whether a profile has been created:

```
Test-Path $PROFILE
```

Or, you can use it in a command to create a profile:

```
New-Item -ItemType file -Path $PROFILE -Force
```

You can use it in a command to open the profile in **notepad.exe**:

```
notepad.exe $PROFILE
```

$PSBoundParameters

Contains a dictionary of the parameters that are passed to a script or function and their current values. This variable has a value only in a scope where parameters are declared, such as a script or function. You can use it to display or change the current values of parameters or to pass parameter values to another script or function.

In this example, the **Test2** function passes the `$PSBoundParameters` to the **Test1** function. The `$PSBoundParameters` are displayed in the format of **Key** and **Value**.

```
function Test1 {
   param($a, $b)
```

```
    # Display the parameters in dictionary format.
    $PSBoundParameters
}

function Test2 {
    param($a, $b)

    # Run the Test1 function with $a and $b.
    Test1 @PSBoundParameters
}
```

```
Test2 -a Power -b Shell
```

```
Key   Value
---   -----
a     Power
b     Shell
```

$PSCmdlet

Contains an object that represents the cmdlet or advanced function that's being run.

You can use the properties and methods of the object in your cmdlet or function code to respond to the conditions of use. For example, the **ParameterSetName** property contains the name of the parameter set that's being used, and the **ShouldProcess** method adds the **WhatIf** and **Confirm** parameters to the cmdlet dynamically.

For more information about the `$PSCmdlet` automatic variable, see about_Functions_CmdletBindingAttribute and about_Functions_Advanced.

$PSCommandPath

Contains the full path and filename of the script that's being run. This variable is valid in all scripts.

$PSCulture

Beginning in PowerShell 7, `$PSCulture` reflects the culture of the current PowerShell runspace (session). If the culture is changed in a PowerShell runspace, the `$PSCulture` value for that runspace is updated.

The culture determines the display format of items such as numbers, currency, and dates, and is stored in a **System.Globalization.CultureInfo** object. Use `Get-Culture` to display the computer's culture. `$PSCulture` contains the **Name** property's value.

$PSDebugContext

While debugging, this variable contains information about the debugging environment. Otherwise, it contains a **null** value. As a result, you can use it to determine whether the debugger has control. When populated, it contains a **PsDebugContext** object that has **Breakpoints** and **InvocationInfo** properties.

The **InvocationInfo** property has several useful properties, including the **Location** property. The **Location** property indicates the path of the script that's being debugged.

$PSEdition

Contains the same value in `$PSVersionTable.PSEdition`. This variable is available for use in module manifest files, whereas `$PSVersionTable` isn't.

$PSHOME

Contains the full path of the installation directory for PowerShell, typically, `C:\Program Files\PowerShell\7` in Windows systems. You can use this variable in the paths of PowerShell files. For example, the following command searches the conceptual Help topics for the word **Help**:

```
Select-String -Pattern Help -Path $PSHOME\en-US\*.txt
```

$PSItem

Same as `$_`. Contains the current object in the pipeline object. You can use this variable in commands that perform an action on every object in a pipeline.

For more information, see about_PSItem.

$PSScriptRoot

Contains the full path of the executing script's parent directory.

In PowerShell 2.0, this variable is valid only in script modules (`.psm1`). Beginning in PowerShell 3.0, it's valid in all scripts.

$PSSenderInfo

Contains information about the user who started the PSSession, including the user identity and the time zone of the originating computer. This variable is available only in PSSessions.

The `$PSSenderInfo` variable includes a user-configurable property, **ApplicationArguments**, that by default, contains only the `$PSVersionTable` from the originating session. To add data to the **ApplicationArguments** property, use the **ApplicationArguments** parameter of the `New-PSSessionOption` cmdlet.

$PSUICulture

Contains the name of the user interface (UI) culture that's configured in the operating system. The UI culture determines which text strings are used for user interface elements, such as menus and messages. This is the value of the **System.Globalization.CultureInfo.CurrentUICulture.Name** property of the system. To get the **System.Globalization.CultureInfo** object for the system, use the `Get-UICulture` cmdlet.

$PSVersionTable

Contains a read-only hash table that displays details about the version of PowerShell that's running in the current session. The table includes the following items:

- **PSVersion** - The PowerShell version number
- **PSEdition** This property has the value of 'Desktop' for PowerShell 4 and below as well as PowerShell 5.1 on full-featured Windows editions. This property has the value of `Core` for PowerShell 6 and higher as well as Windows PowerShell 5.1 on reduced-footprint editions like Windows Nano Server or Windows IoT.
- **GitCommitId** - The commit Id of the source files, in GitHub,
- **OS** - Description of the operating system that PowerShell is running on.
- **Platform** - Platform that the operating system is running on. The value on Linux and macOS is **Unix**. See `$IsMacOs` and `$IsLinux`.
- **PSCompatibleVersions** - Versions of PowerShell that are compatible with the current version
- **PSRemotingProtocolVersion** - The version of the PowerShell remote management protocol.
- **SerializationVersion** - The version of the serialization method
- **WSManStackVersion** - The version number of the WS-Management stack

$PWD

Contains a path object that represents the full path of the current directory location for the current PowerShell runspace.

Note

PowerShell supports multiple runspaces per process. Each runspace has its own *current directory*. This isn't the same as the current directory of the process: `[System.Environment]::CurrentDirectory`.

$Sender

Contains the object that generated this event. This variable is populated only within the Action block of an event registration command. The value of this variable can also be found in the Sender property of the **PSEventArgs** object that `Get-Event` returns.

$ShellId

Contains the identifier of the current shell.

$StackTrace

Contains a stack trace for the most recent error.

$switch

Contains the enumerator not the resulting values of a `Switch` statement. The `$switch` variable exists only while the `Switch` statement is running; it's deleted when the `switch` statement completes execution. For more information, see about_Switch.

Enumerators contain properties and methods you can use to retrieve loop values and change the current loop iteration. For more information, see Using Enumerators.

$this

The `$this` variable is used in script blocks that extend classes to refer to the instance of the class itself.

PowerShell's Extensible Type System (ETS) allows you to add properties to classes using script blocks. In a script block that defines a script property or script method, the `$this` variable refers to an instance of object of the class that's being extended. For example, PowerShell uses ETS to add the **BaseName** property to the **FileInfo** class.

```
PS> Get-ChildItem .\README.md | Get-Member BaseName | Format-List

TypeName   : System.IO.FileInfo
Name       : BaseName
MemberType : ScriptProperty
Definition : System.Object BaseName {get=if ($this.Extension.Length -gt
0)
             {$this.Name.Remove($this.Name.Length -
$this.Extension.Length
             )}else{$this.Name};}
```

For more information, see about_Types.ps1xml.

In a PowerShell class, the `$this` variable refers to the instance object of the class itself, allowing access to properties and methods defined in the class. For more information, see about_Classes.

The `$this` variable is also used by .NET event classes that take script blocks as delegates for the event handler. In this scenario, `$this` represents the object originating the event, known as the event sender.

$true

Contains **True**. You can use this variable to represent **True** in commands and scripts.

Using Enumerators

The `$input`, `$foreach`, and `$switch` variables are all enumerators used to iterate through the values processed by their containing code block.

An enumerator contains properties and methods you can use to advance or reset iteration, or retrieve iteration values. Directly manipulating enumerators isn't considered best practice.

- Within loops, flow control keywords break and continue should be preferred.

- Within functions that accept pipeline input, it's best practice to use parameters with the **ValueFromPipeline** or **ValueFromPipelineByPropertyName** attributes.

  For more information, see about_Functions_Advanced_Parameters.

MoveNext

The MoveNext method advances the enumerator to the next element of the collection. **MoveNext** returns `True` if the enumerator was successfully advanced, `False` if the enumerator has passed the end of the collection.

Note

The **Boolean** value returned by **MoveNext** is sent to the output stream. You can suppress the output by typecasting it to `[void]` or piping it to Out-Null.

```
$input.MoveNext() | Out-Null
```

```
[void]$input.MoveNext()
```

Reset

The Reset method sets the enumerator to its initial position, which is **before** the first element in the collection.

Current

The Current property gets the element in the collection, or pipeline, at the current position of the enumerator.

The **Current** property continues to return the same property until **MoveNext** is called.

Examples

Example 1: Using the $input variable

In the following example, accessing the `$input` variable clears the variable until the next time the process block executes. Using the **Reset** method resets the `$input` variable to the current pipeline value.

```
function Test
{
    begin
    {
        $i = 0
    }

    process
    {
        "Iteration: $i"
        $i++
        "`tInput: $input"
        "`tAccess Again: $input"
        $input.Reset()
        "`tAfter Reset: $input"
    }
}

"one","two" | Test
```

```
Iteration: 0
    Input: one
    Access Again:
    After Reset: one
Iteration: 1
    Input: two
    Access Again:
    After Reset: two
```

The process block automatically advances the `$input` variable even if you don't access it.

```
$skip = $true
function Skip
{
    begin
    {
        $i = 0
    }

    process
    {
        "Iteration: $i"
        $i++
        if ($skip)
        {
            "`tSkipping"
            $skip = $false
        }
        else
        {
            "`tInput: $input"
        }
    }
}

"one","two" | Skip
```

```
Iteration: 0
    Skipping
Iteration: 1
    Input: two
```

Example 2: Using $input outside the process block

Outside of the process block the `$input` variable represents all the values piped into the function.

- Accessing the `$input` variable clears all values.
- The **Reset** method resets the entire collection.

- The **Current** property is never populated.
- The **MoveNext** method returns false because the collection can't be advanced.
    - Calling **MoveNext** clears out the `$input` variable.

```
Function All
{
    "All Values: $input"
    "Access Again: $input"
    $input.Reset()
    "After Reset: $input"
    $input.MoveNext() | Out-Null
    "After MoveNext: $input"
}

"one","two","three" | All
```

```
All Values: one two three
Access Again:
After Reset: one two three
After MoveNext:
```

Example 3: Using the $input.Current property

With the **Current** property, the current pipeline value can be accessed multiple times without using the **Reset** method. The process block doesn't automatically call the **MoveNext** method.

The **Current** property is never populated unless you explicitly call **MoveNext**. The **Current** property can be accessed multiple times inside the process block without clearing its value.

```
function Current
{
    begin
    {
        $i = 0
    }

    process
    {
        "Iteration: $i"
        $i++
        "`tBefore MoveNext: $($input.Current)"
        $input.MoveNext() | Out-Null
        "`tAfter MoveNext: $($input.Current)"
        "`tAccess Again: $($input.Current)"
    }
}
```

```
"one","two" | Current
```

```
Iteration: 0
    Before MoveNext:
    After MoveNext: one
    Access Again: one
Iteration: 1
    Before MoveNext:
    After MoveNext: two
    Access Again: two
```

Example 4: Using the $foreach variable

Unlike the `$input` variable, the `$foreach` variable always represents all items in the collection when accessed directly. Use the **Current** property to access the current collection element, and the **Reset** and **MoveNext** methods to change its value.

Note

Each iteration of the `foreach` loop automatically calls the **MoveNext** method.

The following loop only executes twice. In the second iteration, the collection is moved to the third element before the iteration is complete. After the second iteration, there are now no more values to iterate, and the loop terminates.

The **MoveNext** property doesn't affect the variable chosen to iterate through the collection (`$Num`).

```
$i = 0
foreach ($num in ("one","two","three"))
{
    "Iteration: $i"
    $i++
    "`tNum: $num"
    "`tCurrent: $($foreach.Current)"

    if ($foreach.Current -eq "two")
    {
        "Before MoveNext (Current): $($foreach.Current)"
        $foreach.MoveNext() | Out-Null
        "After MoveNext (Current): $($foreach.Current)"
        "Num hasn't changed: $num"
    }
}
```

```
Iteration: 0
        Num: one
        Current: one
Iteration: 1
        Num: two
        Current: two
Before MoveNext (Current): two
After MoveNext (Current): three
Num hasn't changed: two
```

Using the **Reset** method resets the current element in the collection. The following example loops through the first two elements *twice* because the **Reset** method is called. After the first two loops, the `if` statement fails and the loop iterates through all three elements normally.

Important

This could result in an infinite loop.

```
$stopLoop = 0
foreach ($num in ("one","two", "three"))
{
    ("`t" * $stopLoop) + "Current: $($foreach.Current)"

    if ($num -eq "two" -and $stopLoop -lt 2)
    {
        $foreach.Reset()
        ("`t" * $stopLoop) + "Reset Loop: $stopLoop"
        $stopLoop++
    }
}
```

```
Current: one
Current: two
Reset Loop: 0
        Current: one
        Current: two
        Reset Loop: 1
                Current: one
                Current: two
                Current: three
```

Example 5: Using the $switch variable

The `$switch` variable has the exact same rules as the `$foreach` variable. The following example demonstrates all the enumerator concepts.

Note

Note how the **NotEvaluated** case is never executed, even though there's no `break` statement after the **MoveNext** method.

```
$values = "Start", "MoveNext", "NotEvaluated", "Reset", "End"
$stopInfinite = $false
switch ($values)
{
    "MoveNext" {
        "`tMoveNext"
        $switch.MoveNext() | Out-Null
        "`tAfter MoveNext: $($switch.Current)"
    }
    # This case is never evaluated.
    "NotEvaluated" {
        "`tAfterMoveNext: $($switch.Current)"
    }

    "Reset" {
        if (!$stopInfinite)
        {
            "`tReset"
            $switch.Reset()
            $stopInfinite = $true
        }
    }

    default {
        "Default (Current): $($switch.Current)"
    }
}
```

```
Default (Current): Start
    MoveNext
    After MoveNext: NotEvaluated
    Reset
Default (Current): Start
    MoveNext
    After MoveNext: NotEvaluated
Default (Current): End
```

See also

- about_Functions
- about_Functions_Advanced
- about_Functions_Advanced_Methods
- about_Functions_Advanced_Parameters
- about_Functions_OutputTypeAttribute
- about_Functions_CmdletBindingAttribute
- about_Hash_Tables
- about_Preference_Variables

- about_Splatting
- about_Variables

In this article

Previous Chapter Next Chapter